INTRODUCTION TO BASIC PART II

commodore COMPUTER

CONTENTS

This course is Part 2 of a series designed to help you learn about every aspect of programming the Commodore 64 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 64 computer. The course has two constituent parts:

1. A self-study text divided into 11 lessons or 'units', each of which deals with an important aspect of programming.

2. Two cassette tapes or one diskette containing a collection of 64 programs, which will help you study the units.

CONTENTS LIST

INTRODUCTION

Welcome to the second part of the COMMODORE 64 BASIC course. The layout of the book and tapes or diskettes will be familiar to you since the course is a direct continuation of 64 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 or diskettes 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

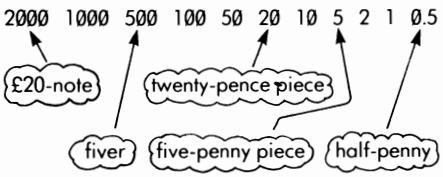
This unit introduces an important and useful group of commands which use the keywords DATA, READ and RESTORE.

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 COMMODORE 64 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 64 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 64, and who has his own private keyboard. Every time the 64 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
     NOTE(S)
1010 DATA 1000, TEN-POUND NOTE
1020 DATA  500, FIVE-POUND NOTE
1030 DATA  100, ONE-POUND COIN(S)
1040 DATA   50, FIFTY-PENNY PIECE
1050 DATA   20, TWENTY-PENNY PC(S)
1060 DATA   10, TEN-PENNY PIECE
1070 DATA    5, FIVE-PENNY PIECE
1080 DATA    2, TWOPENCE(S)
1090 DATA    1, PENNY
1100 DATA  0.5, HALFPENNY
```


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
$1     DOLLAR(S)
$0.25  QUARTER(S)
$0.10  DIME(S)
$0.05  NICKEL
$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 2 screen lines (80 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 64 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:

```
1000 DATA 2000
1010 DATA TWENTY-POUND NOTE(S)
1030 DATA 1000
1040 DATA TEN-POUND NOTE
```

or

```
1000 DATA 2000, TWENTY-POUND
NOTES, 1000, TEN-POUND
NOTE,500,FIVE-POUND NOTE,100, . . .
```

or even

```
1000 DATA 2000
1010DATA TWENTY-POUND NOTE(S),1000
1020 DATA TEN-POUND NOTE,500,
     FIVE-POUND NOTE,100
1030 DATA ONE-POUND COIN(S)
. . .
```

In other words, the 64 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 64 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:

```
 10 READ A
. . .
. . .
100 DATA HELLO
```

you will get

```
? SYNTAX
ERROR IN 100
```

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

```
10 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 64 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

```
1000 DATA JANUARY,FEBRUARY,MARCH,
     APRIL
1010 DATA MAY,JUNE,JULY,AUGUST
1030 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

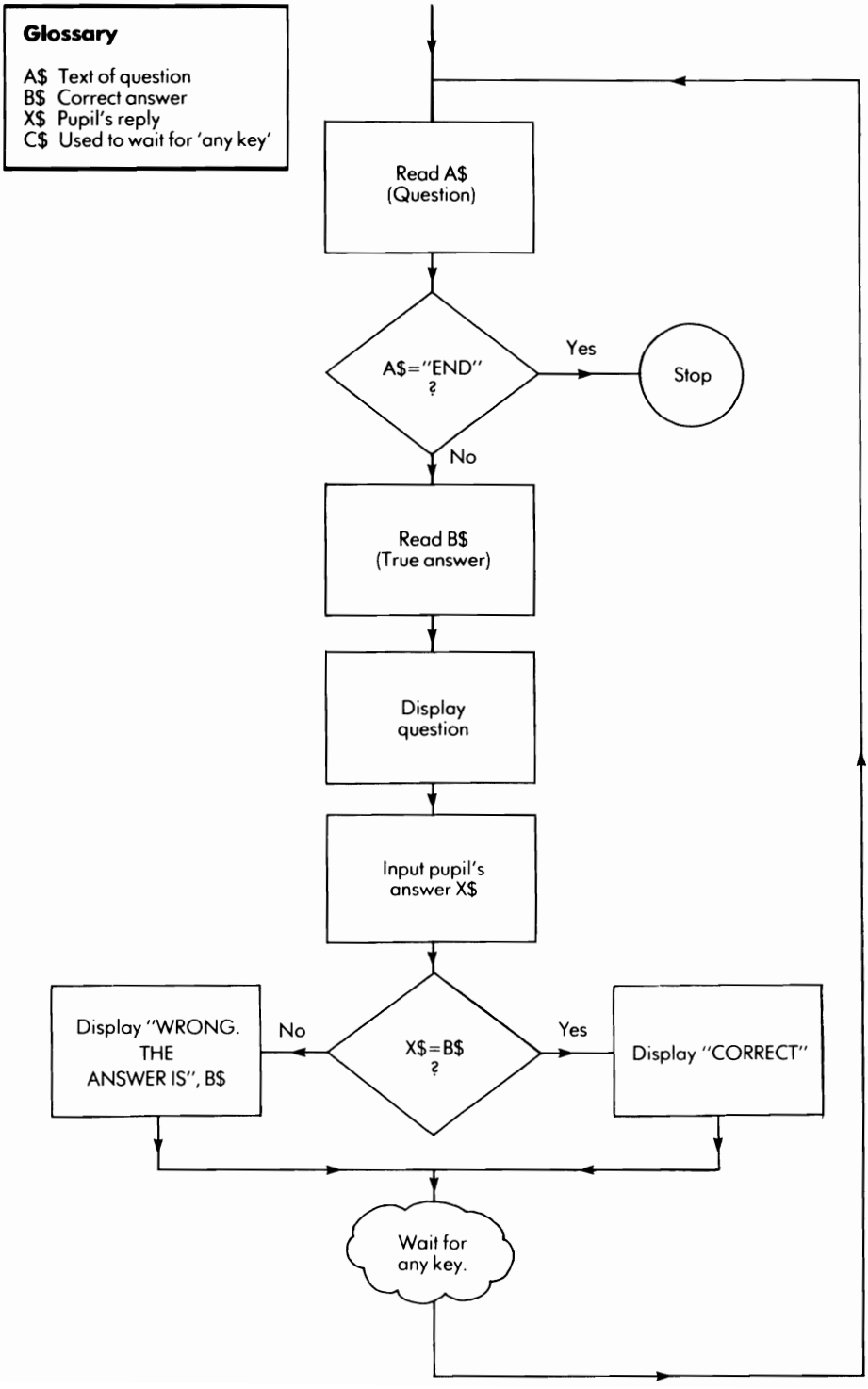
Use the same DATA statements as in the previous program. Write your program in the form of a loop including RESTORE so that when one date has been displayed another can be input.

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 64 and try it out:

```
  10 READ A$
  20 IF A$ = "END" THEN 190
  30 READ B$
  40 PRINT " SHIFT and CLR HOME "
  50 PRINT A$
  60 PRINT
  70 INPUT X$
  80 PRINT
  90 IF X$ = B$ THEN 130
 100 PRINT "WRONG. THE ANSWER IS"
 110 PRINT B$
 120 GOTO 140
 130 PRINT "CORRECT"
 140 PRINT
 150 PRINT "NOW PRESS ANY KEY"
 160 GET C$
 170 IF C$ = "" THEN 160
 180 GOTO 10
 190 STOP
 500 DATA WHAT IS THE CAPITAL OF
     FRANCE,PARIS
 510 DATA WHAT COUNTRY HAS TOKYO
     AS ITS CAPITAL,JAPAN
 520 DATA WHAT COUNTRY HAS THE
     LARGEST POPULATION,CHINA
 530 DATA WHAT COUNTRY LIES DIRECTLY
     SOUTH OF THE USA,MEXICO
1000 DATA END
```


Glossary
A$ Text of question
B$ Correct answer
X$ Pupil's reply
C$ Used to wait for 'any key'
Read A$ (Question)
A$="END" ?
Yes
Stop
No
Read B$ (True answer)
Display question
Input pupil's answer X$
X$=B$?
No
Display "WRONG. THE ANSWER IS", B$
Yes
Display "CORRECT"
Wait for any key.

When you have got the program running, make up some more questions and add them in, using label numbers 540 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 UNIT16QUIZ64.

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:

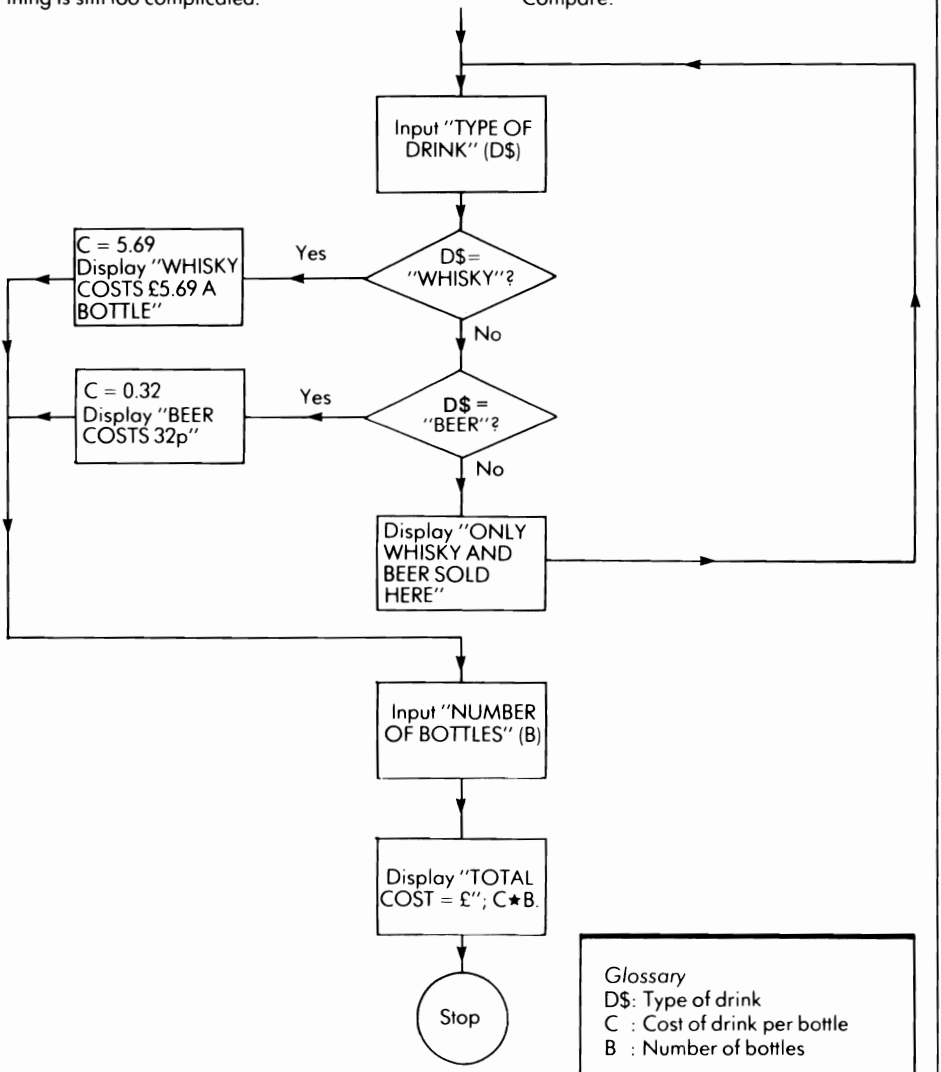


Figure 17.1

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 COMMODORE 64, 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 64's internal line length of 80 characters (2 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 64 always immediately 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 64 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

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
```

c) Load the program entitled UNIT17PROG64 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:

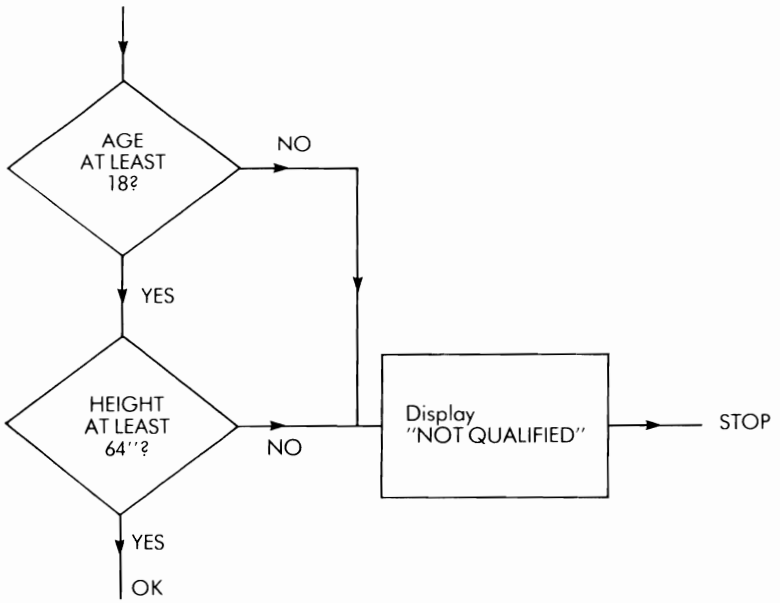


Figure 17.2

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 80 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
```

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 64 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) Use De Morgan's Laws to express the following compound conditions without using NOTs:

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

b) 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!!

c) 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

INTRODUCING SUBROUTINES

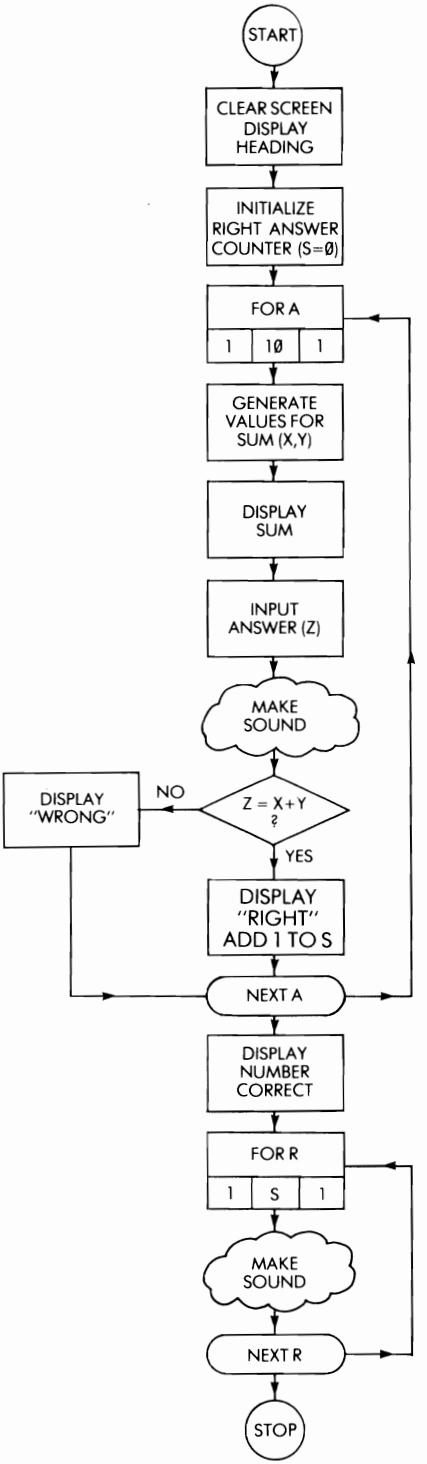
Up to this point in the course, you have gained plenty of practice in writing small programs — say up to 30 commands or so. Sooner or later, you are bound to discover that most interesting programs have to be very much longer, and that you need specialised tools and techniques to help you build them 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.

The program presents you with 10 simple addition sums made up at random. It records whether your answers were right or wrong and ends by giving you a score out of ten.



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:

```
10 VV = 212*256
20 POKE VV+24,15: POKE VV,0
30 POKE VV+1, 80: POKE VV+5,9
40 POKE VV+4,0: POKE VV+4,33
50 FOR M=1 TO 800: 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:

```
1000 REM SUBROUTINE TO MAKE PIP
     SOUND
1010 VV = 212*256
1020 POKE VV + 24, 15: POKE VV,0
1030 POKE VV+1,80: POKE VV+5,9
1040 POKE VV+4,0 : POKE VV+4,33
1050 FOR MM = 1 TO 800 : NEXT MM
1060 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 1000

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:

```
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 IF Z=X+Y THEN 130
110 PRINT "WRONG"
120 GOTO 150
130 PRINT "RIGHT"
140 S=S+1
150 NEXT A
155 FOR T =1 TO 500:NEXT T
160 PRINT "THAT'S";S;"RIGHT OUT OF
    10"
170 FOR R=1 TO S
180 GOSUB 1000: REM MAKE SOUND
190 NEXT R
200 STOP
1000 REM SUBROUTINE TO MAKE PIP
     SOUND
1010 VV = 212*256
1020 POKE VV + 24, 15: POKE VV,0
1030 POKE VV+1,80: POKE VV+5,9
1040 POKE VV+4,0 : POKE VV+4,33
1050 FOR MM = 1 TO 800 : NEXT MM
1060 RETURN
```

HOW GOSUB WORKS

Let's follow the program through. The GOSUB is very like a GOTO, but with one important difference. When the 64 jumps to the subroutine, it remembers the location 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 location number stored in the stack! This provides an automatic mechanism which ensures that whenever the 64 finishes executing a subroutine, it always gets back to the right place in the main program. The location stored in the stack while the 64 is obeying the subroutine is called a *link* or *return address*.

Our program begins by obeying the commands in line 10 in the ordinary way. Command 90 says GOSUB 1000; so the computer jumps to 1000 but on the way it notes the next instruction which in this case is a REM statement. The address location of this REM statement is put into the stack. This is the link or return address.

Once it reaches the subroutine the machine executes the commands 1010-1050 the last line of which is RETURN. Since the stack contains the address of the REM statement in line 90 the RETURN sends the machine back to this point in the program. As this is only a REM statement the machine proceeds to the next line.

When all the sums have been answered the program comes to another subroutine call, line 180. It again jumps to line 1000, but this time the link address placed in the stack is for the REM statement in line 180. When RETURN is obeyed the second time, control returns to the REM statement in line 180, not 90 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 176 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

★★★★★★★★★★★

```
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
```

Let's examine this routine closely. Commands 3010 to 3030 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 3000
```

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:

```
10 FOR X1=1 TO 18
20 GOSUB 3000
30 NEXT X1
40 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 3000 and to use X1 as the parameter. At this stage, he could equally well have put the subroutine anywhere else (say 4500) 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

```
3000 REM .....
```

Calls had to use X1 as parameter and be written as

```
GOSUB 3000
```


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.

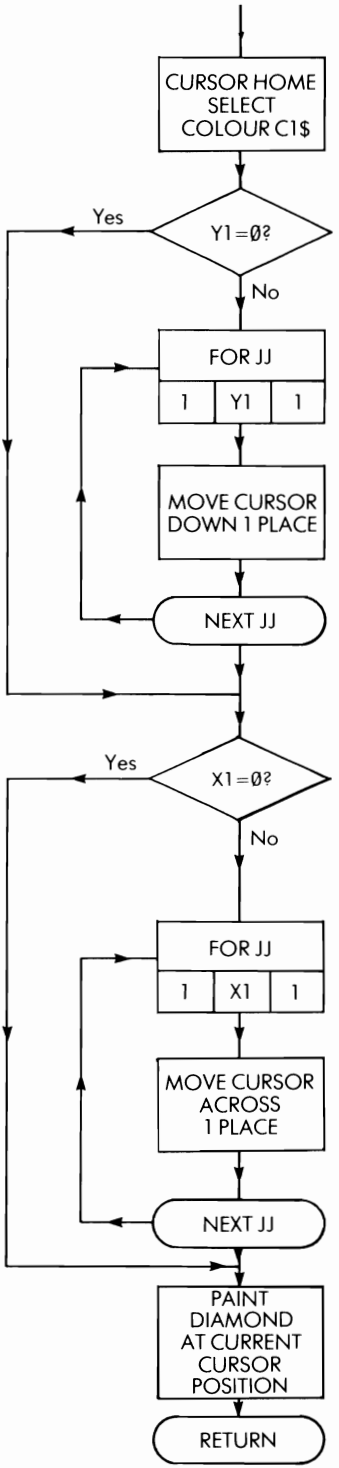
USING MORE THAN ONE PARAMETER

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:

```
2000 REM DISPLAY C1$-COLOURED
     DIAMOND AT X1 SPACES ACROSS
     AND Y1 PLACES DOWN
2010 PRINT " [CLR HOME] ";C1$;
2020 IF Y1=0 THEN 2060
2030 FOR JJ=1 TO Y1
2040 PRINT " [CRSR ↓] ";
2050 NEXT JJ
2060 IF X1 = 0 THEN 2100
2070 FOR JJ=1 TO X1
2080 PRINT " [CRSR →] ";
2090 NEXT JJ
2100 PRINT " [CTRL] and [RVS ON]
     [CTRL] and [RVS OFF] [CRSR ↓]
     [SHIFT] and [CRSR ←] [SHIFT] and [CRSR →]
     ";
2110 RETURN
```

This subroutine uses cursor commands in strings to move the cursor round the screen. 2100 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 64 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=0 is included so that the FOR loop in lines 2030-2050 can be skipped altogether if necessary. The test for X1=0 is there for a similar reason.

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

```
10 PRINT " [SHIFT] and [CLR HOME] ";
20 C1$= " [CTRL] and [GRN] "
30 FOR X1=1 TO 37 STEP 3
40 FOR Y1=0 TO 21 STEP 3
50 GOSUB 2000: REM DRAW C1$-
   COLOURED DIAMOND AT X1,Y1
60 NEXT Y1
70 NEXT X1
80 GOTO 80: REM LOOP STOP
```

EXPERIMENT 18·2

Write a subroutine, starting at line 500, 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:

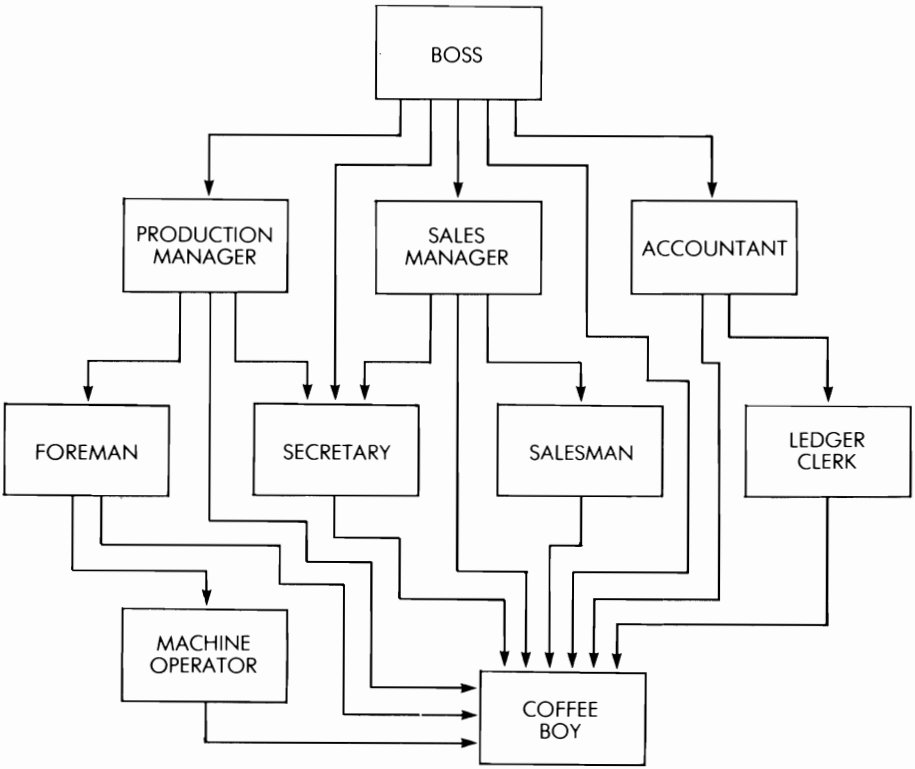
```
10 PRINT " [SHIFT] and [CLR HOME] ";
20 FOR X1 = 0 TO 35
30 C1$= " [CTRL] and [RED] "
40 GOSUB 500: REM DRAW RED MONSTER
   AT X1
50 FOR T=1 TO 150: NEXT T: REM WAIT A
   BIT
60 C1$= " [CTRL] and [BLU] "
70 GOSUB 500: REM ERASE MONSTER BY
   DRAWING IT IN BLUE
80 NEXT X1
90 GOTO 90: REM LOOP STOP
```

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. The lowly coffee boy is clearly the most important person in the office, since all the other people depend on him!

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

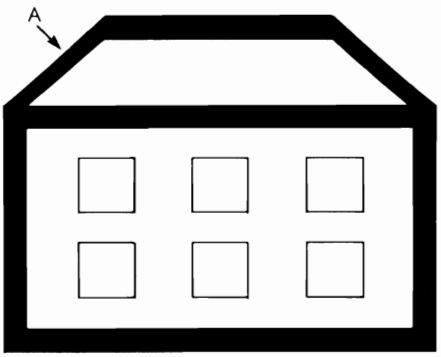


This type of structure can be reflected in a BASIC program. A subroutine may be called by the main program or it may be called by other subroutines, to at the most twenty-four 'levels'. The computer doesn't get mixed up because it makes special arrangements to store all the links it needs to get back 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.

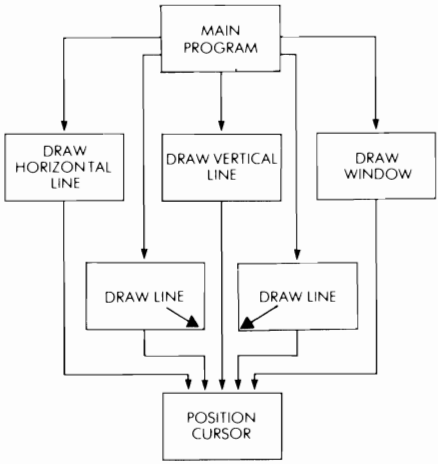
EXPERIMENT 18·3

Load and run the program called PICTURE **64** from the cassette tape or diskette. 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 **1000** 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 **2000** draws lines downwards, the subroutine at **3000** diagonally upwards from left to right, and the one at **4000**, diagonally downwards from left to right. Thus to draw the line marked A in the picture, the calling sequence is

X1=11: Y1=8: N1=5: GOSUB 3000



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 10 to 140, 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

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:

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

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 10 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 5500. 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: 5500 to ☐

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

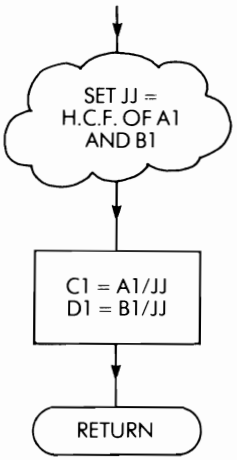
Local Variables: ☐

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 10 and 24 is 2, and 10/24 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

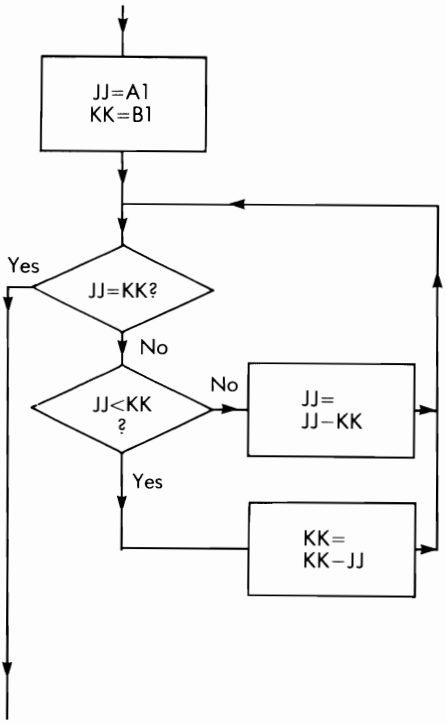
A1/B1: Fraction to be reduced to its lowest terms
C1/D1: Result

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: 5500 to 5570

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 5500
(c) To look for the results in C1 and D1
(e) To avoid using lines 5500 to 5570 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:

```
10 INPUT "FRACTION"; A1,B1
20 GOSUB 5500
30 PRINT "RESULT ="; C1; "/"; D1
40 GOTO 10
```

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? 10, 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: $p/q+s/t = \dfrac{p\star t + q\star s}{q\star t}$

eg:

$3/8+5/12 = \dfrac{3\star 12+5\star 8}{8\star 12}$

Set A1 = P★T + Q★S and B1 = Q★T, 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 178 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:

39: Works correctly.

40: Gives 40 stars and an extra blank line! If we were using the subroutine to draw a chart, this would spoil its appearance.

41: This *doesn't* give a line with 40 stars; it displays a line of 40 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:

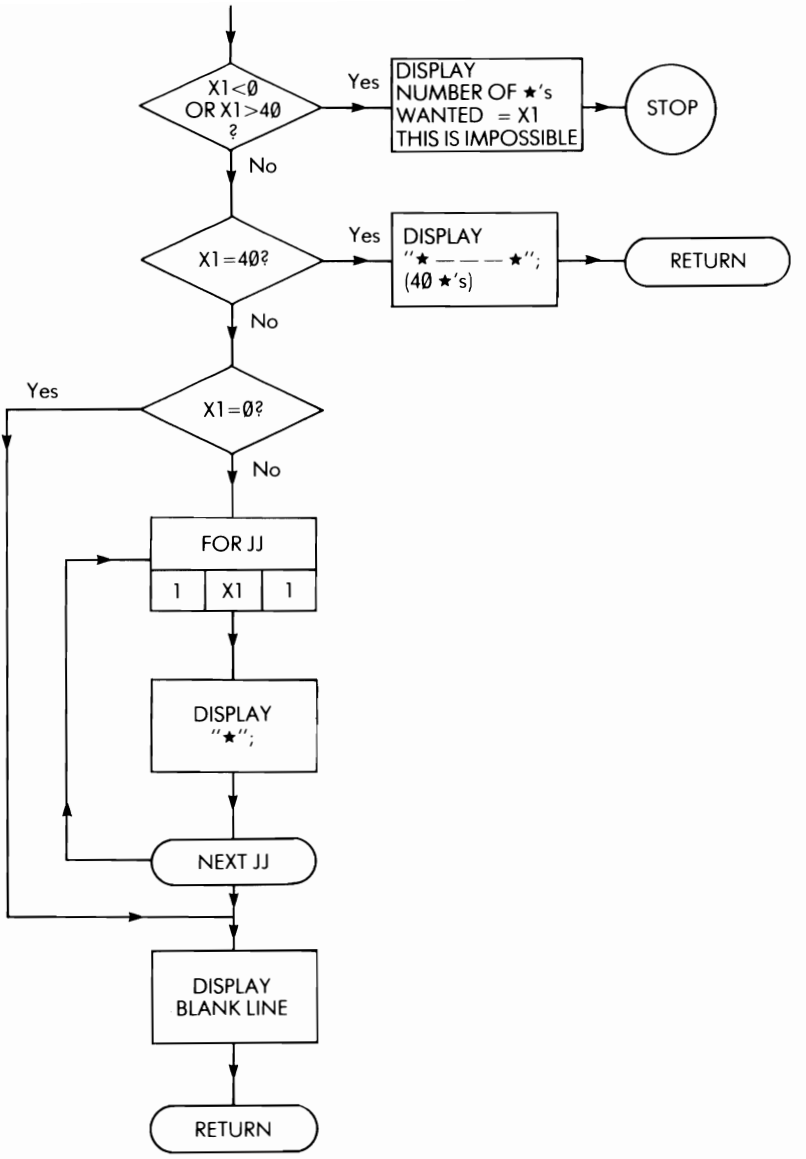
First, a line can only hold between 0 and 40 stars, so we should make the subroutine *reject* any value outside this range. A calling program which did supply an out-of-range value for the parameter X1 would presumably be in error, so it is appropriate to make the subroutine display a warning message and stop.

Second, the subroutine needs special provision for the extreme values 0 and 40. 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, 40 led to an unwanted extra line because a new line is *automatically* inserted after the 40th 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.
3005 REM MUST BE IN RANGE 0-40.
     LOCAL IS JJ.
3010 IF X1<0 OR X1>40 THEN PRINT "NO.
     OF ★'S WANTED=";X1:
     GOTO 3090
3020 IF X1=40 THEN 3100
3030 IF X1=0 THEN 3090
3040 FOR JJ=1 TO X1
3050 PRINT "★";
3060 NEXT JJ:PRINT:RETURN
3090 PRINT"THIS IS IMPOSSIBLE":STOP
3100 PRINT"★★★★★★★★★★★★★★★★★★★★
★★★★★★★★★★★★★★★★★★★★";: RETURN
```

A main driver program for this subroutine is:

```
10 INPUT "NUMBER OF STARS"; X1
20 GO SUB 3000
30 GOTO 10
```

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 40) 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 batting 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 run. Run the program both with the old (page 188) 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, 7, 25, 40, 5, 0, 4, 1, 0, 40, 15
```

(b) Go back to the subroutine given on page 186, 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 (on page 186), 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	

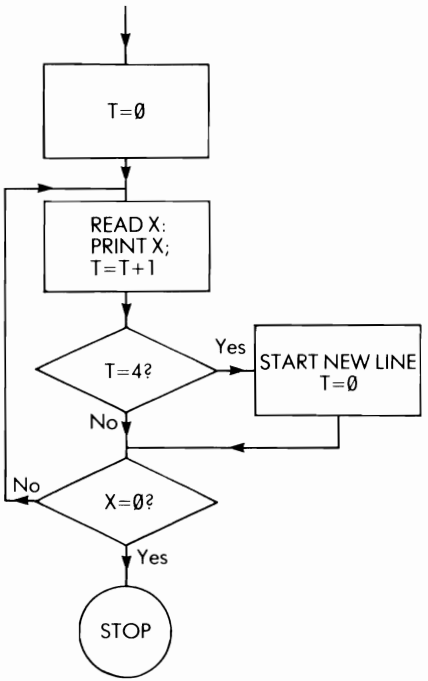
NAMING CONVENTIONS IN SUBROUTINES

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 statements. 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 SUBROUTINE TO MAKE PIP
     SOUND
2Ø1Ø VV = 212★256
2Ø2Ø POKE VV+4,Ø
2Ø3Ø POKE VV+24, 15: POKE VV, Ø
2Ø4Ø POKE VV+1, 8Ø : POKE VV+5, 9
2Ø5Ø POKE VV+4, 33
2Ø6Ø FOR T=1 TO 5ØØ: NEXT T
2Ø7Ø POKE VV+4,Ø
2Ø8Ø 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:

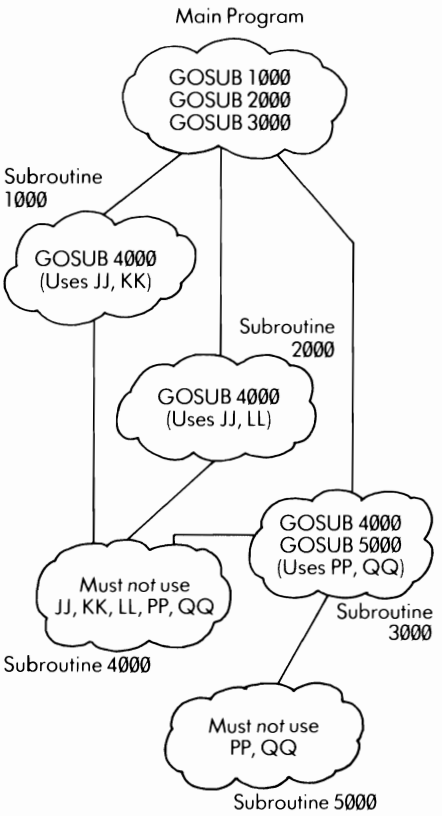
15 GOSUB 2000
and 45 GOSUB 2000.

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)

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 "BIGLETTERS64" on the diskette or 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 64 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 variables A$,B$,C$ and D$, and write the following:

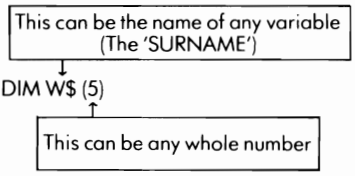
```
10 PRINT "ENTER NAMES OF PUPILS"
20 INPUT A$
30 INPUT B$
40 INPUT C$
50 INPUT D$
60 PRINT "REVERSED ORDER IS"
70 PRINT D$
80 PRINT C$
90 PRINT B$
100 PRINT A$
110 STOP
```

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 tells the 64 to set aside space for a family of string variables called W$. There are *six* of them, and their full names are:

W$(0)
W$(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 0, 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 0, 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(20):REM DECLARES AN ARRAY OF
                21 ELEMENTS CALLED N(0)
                TO N(20)
N(5)=N(3) +5
PRINT N(1); N(2); N(3)
INPUT N(2)
IF N(12) = N(17) THEN 150
```

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
...
NEXT N(5)
```

} Is *not* allowed

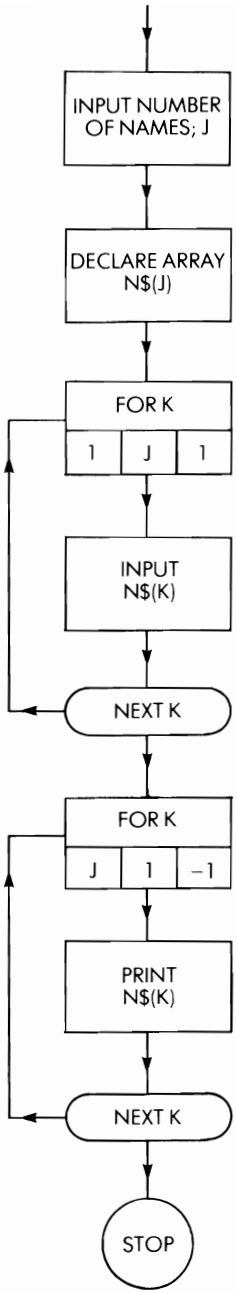
So far, we haven't said anything that would seem to help with the problem of inverting the name list. Here is the key point:

> 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:

```
10 INPUT "HOW MANY NAMES";J
20 DIM N$(J)
30 FOR K=1 TO J
40 INPUT N$(K)
50 NEXT K
60 FOR K=J TO 1 STEP -1
70 PRINT N$(K)
80 NEXT K
90 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$(0)), 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

```
30 DIM A(50)
...
70 DIM A(50)
```

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

```
90 GOTO 10
```

it would give a fault the second time it tried to obey the DIM command in line 20.

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
```

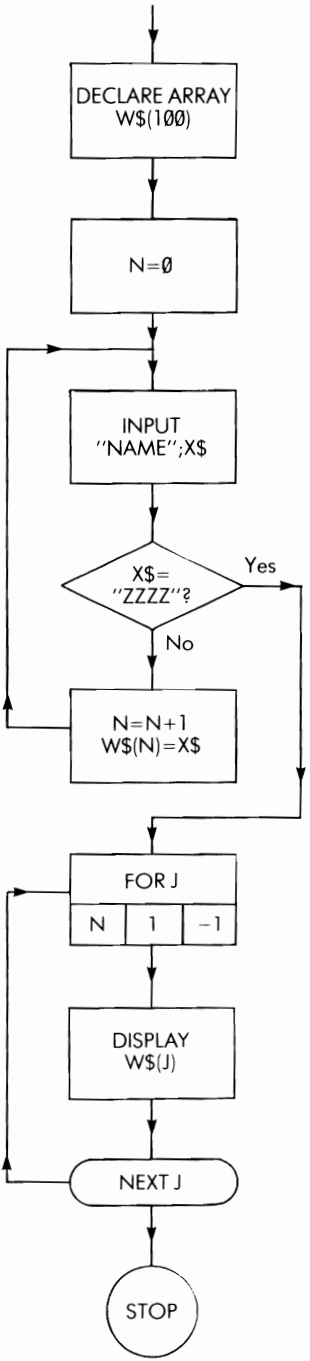
the output would be

```
TURNER
MICHAELANGELO
DURER
CRANACH
```

Hint: Use the following flow chart:

Glossary

W$(100): Array for names (max 100)
N: Count of names
X$: 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 = 0 TO 10
110 PRINT T$(2);" * ";T$(J);" = ";T$(2*J)
120 NEXT J
```

If you key in these instructions following the ones labelled 10-90, 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 0-20

Line numbers: 1000-1060

Parameters: Input: A1$ Word to be converted
Output B1: Value of number

Locals: JJ

External Reference: T$(0-20): Names of the numbers

```
1000 REM CONVERT WORD A1$ INTO
     NUMBER B1
1010 FOR JJ=0 TO 20
1020 IF A1$ = T$(JJ) THEN 1050
1030 NEXT JJ
1040 PRINT "NO ENTRY FOUND": STOP
1050 B1=JJ
1060 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$(0),T$(1), and so on. When it finds a corresponding entry it jumps out of the loop to command 1050. 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:

```
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
100 PRINT "GIVE TWO NUMBERS"
110 INPUT X$,Y$
120 REM SET UP PARAMETERS AND CALL
    SUBROUTINE TO CONVERT X$ TO X
125 A1$ =X$: GOSUB 1000:X=B1
130 REM SAME FOR Y
135 A1$=Y$: GOSUB 1000:Y=B1
140 Z=X+Y: REM ADD THE TWO NUMBERS
150 IF Z > 20 THEN PRINT "RESULT NOT IN
    LIST": STOP
160 PRINT "SUM IS "; T$(Z)
170 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 150 is included because the program can't display any number higher than 20. 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 64 would come up with:

```
? BAD SUBSCRIPT
  ERROR IN 160
```

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 (40).
b) The data statements of a program contains 20 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

NAME? FREUD
FREUD HAS NO LISTED
PHONE NUMBER
```

Typed by User

Experiment 20.2 Completed	

The self test quiz for this unit is entitled UNIT20QUIZ64.

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 64.

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 64 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 64 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 2*nd* 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
```

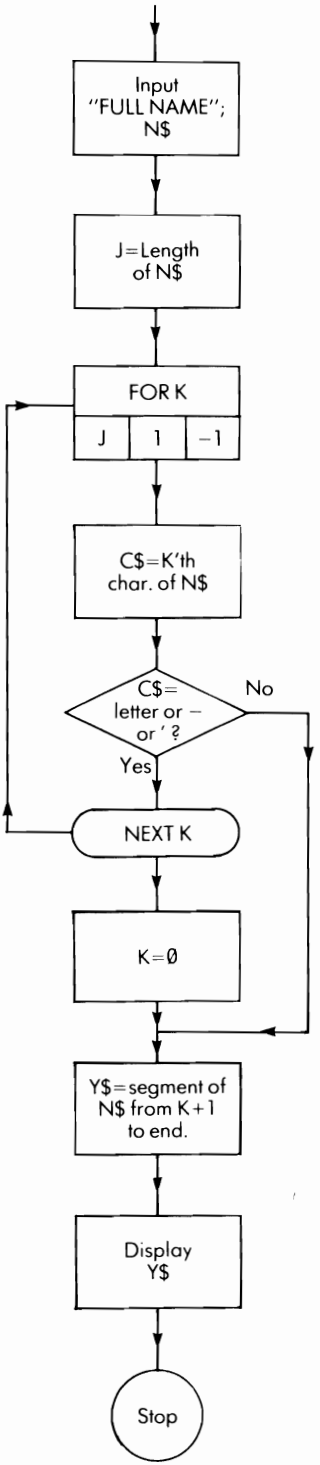
If you search from the right, the first character you come to which can't be part of a surname is the full stop in position 4. The surname therefore starts at position 5, and can be extracted by a MID$ function:

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 (J−K).

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:

```
10 INPUT "FULL NAME"; N$
20 J=LEN(N$)
30 FOR K=J TO 1 STEP -1
40 C$=MID$(N$,K,1)
50 IF NOT(C$>="A"AND C$<="Z"
   OR C$="-"OR C$=" ")THEN 80
60 NEXT K
70 K=0
80 Y$=MID$(N$,K+1,J-K)
90 PRINT Y$
100 GOTO 10
```


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 80 if C$ is *not* part of a surname.

Line 70 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 0. Therefore K must be set so that line 80 can be obeyed.

The command in line 80 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: 4100-4180

Input parameter: N1$ contains a full name

Output parameter: Y1$ delivers the surname

Local variables: JJ, KK, CC$

```
4100 REM EXTRACT SURNAME FROM N1$
     AND DELIVER IT IN Y1$
4110 JJ=LEN (N1$)
4120 FOR KK=JJ TO 1 STEP -1
4130 CC$=MID$(N1$,KK,1)
4140 IF NOT (CC$>="A" AND CC$<="Z"
     OR CC$="-" OR CC$=" ") THEN
     4170
4150 NEXT KK
4160 KK=0
4170 Y1$=MID$(N1$,KK+1,JJ-KK)
4180 RETURN
```

A driver program to test out this subroutine would be:

```
10 INPUT "NAME PLEASE"; N1$
20 GOSUB 4100
30 PRINT "SURNAME IS "; Y1$
40 GOTO 10
```

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$= MID$(X$,1,3) + "U" + MID$(X$,5,LEN(X$)-4)
```

First 3 characters of X$

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 < 24 times > CRSR "

20 X$ = " CRSR < 39 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:

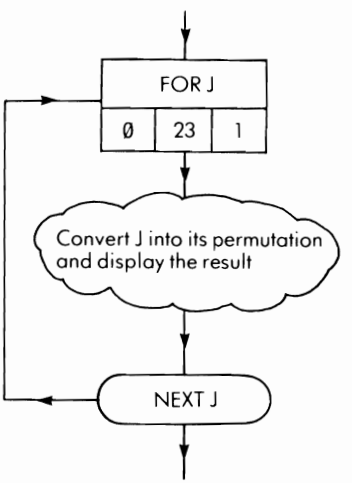
Number of Objects	*Number of Permutations*
1	1
2	2 = 2
3	3x2 = 6
4	4x3x2 = 24
5	5x4x3x2 = 120
6	6x5x4x3x2 = 720
7	7x6x5x4x3x2 = 5040
8	8x7x6x5x4x3x2 = 40320

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:

$$\begin{pmatrix}ABC\\ACB\end{pmatrix}\ \begin{pmatrix}BAC\\BCA\end{pmatrix}\ \begin{pmatrix}CAB\\CBA\end{pmatrix}$$

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.

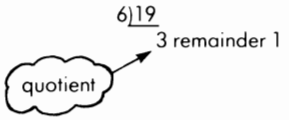
For any permutation, we can define a 'group number' and a 'sub-permutation number'. The group number will indicate the first letter (according to the code A=Ø, B=1, C=2, and so on) and the sub-permutation number will be the position within the group (also starting at Ø). For instance, consider the permutation 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.



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

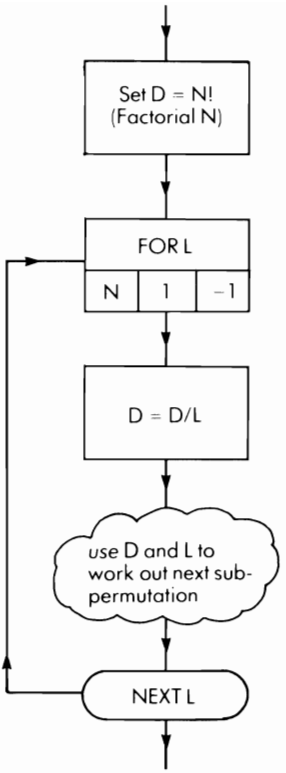
Once a letter has been used, it must be removed from the string. The others will then be moved up automatically, and this is equivalent to relabelling them. Taking a letter out of a string is quite easy: we concatenate (join together) the portion of the string to the left of the used letter and the portion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

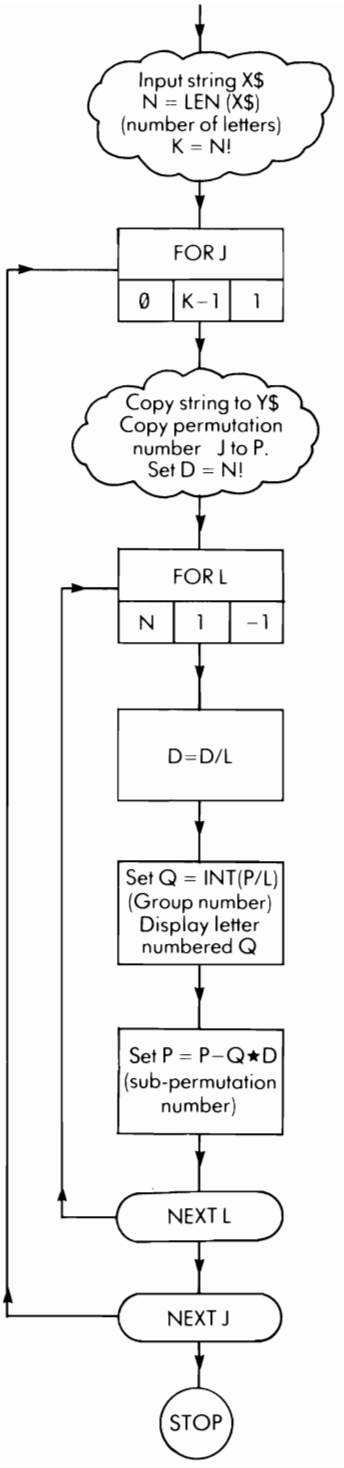


In this flow chart L and D will take the right sequence of values. For instance if N=6, then L will become 6,5,4,3,2,1 and D (by the time it is used) will be 120,24,6,2,1 and 1).

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$: String to be permuted
N: Length of string to be permuted
K: Factorial N (calculated in line 30)
J: Permutation number
Y$: 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 = 17-12 = 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, if 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.

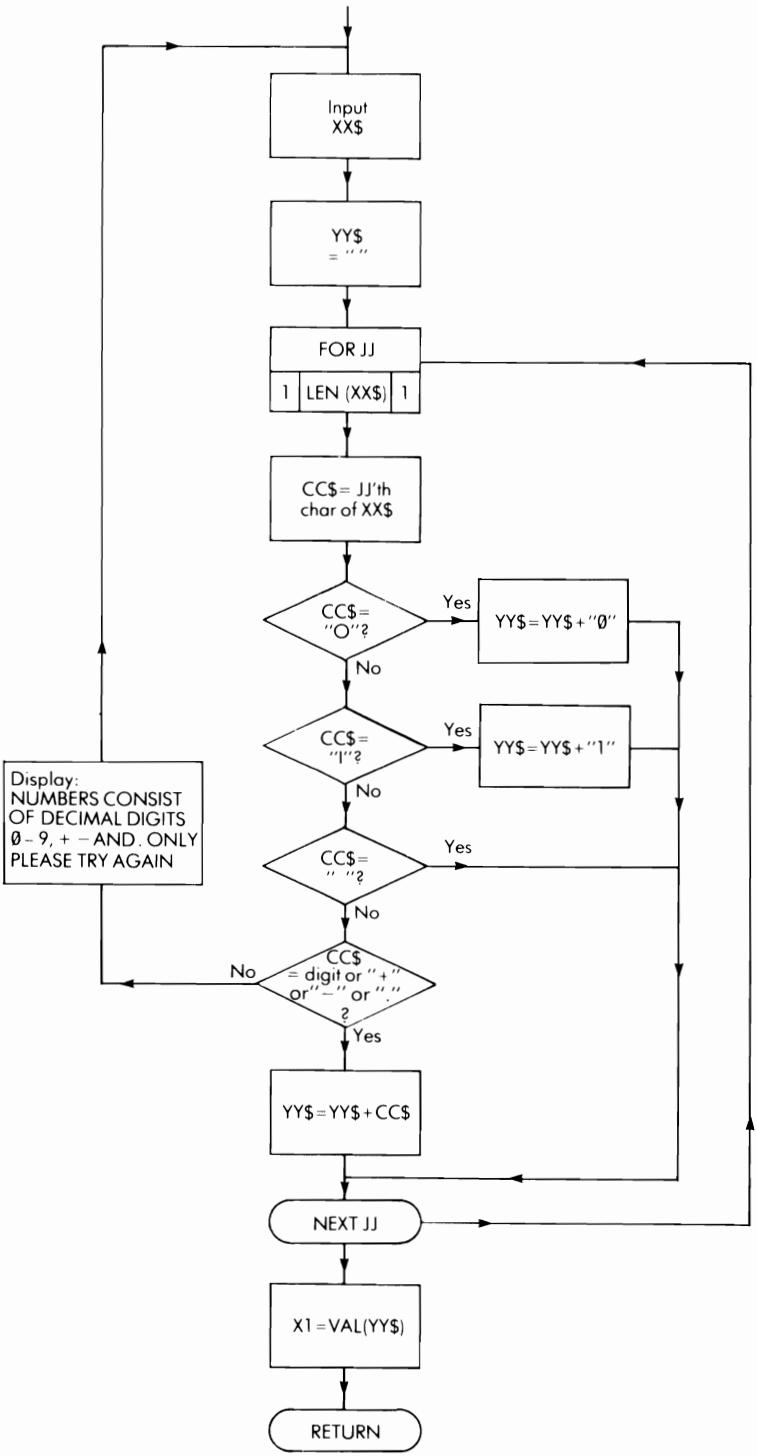
Subroutine Specification

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$, CC$


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

Remember to include the blank strings ("") in lines 5 and 20, otherwise the example won't run the way we expect it to.

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

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	15.999999
9	
4.09999999	16.809999
9	

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 64 — 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 upsets the neat layout of the table.

The second problem emerges when the truncation error affects the printed value of J itself. "3.7" is turned into "3.69999999". This number is so long that it spills out of the space allocated to it on the screen and displaces the second number on the line to the right. When the second number also shows a truncation error, part of it runs over on to the next line, making the table almost unreadable.

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 64 generally displays fractions to 8 decimal places, except that it leaves off trailing zeros. Usually 3 or 4 places are sufficiently accurate 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* 0.0005, and then *throw away* the fourth and subsequent places. These examples show the process at work:

3.14159	2.71818
0.0005 +	0.0005 +
3.14209	2.71878
(discard 09)	(discard 78)
3.142	2.718

It is clear that rounding will get rid of the truncation errors introduced by the 64 (since 3.69999999 rounded to 3 places is 3.700) 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 0.0005

(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:

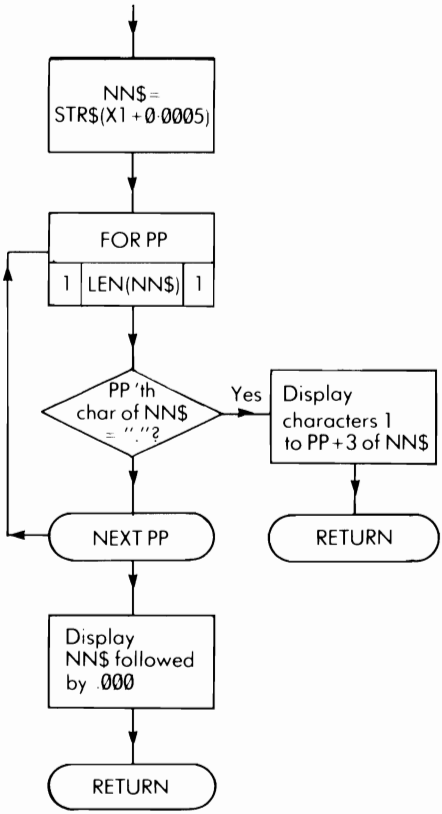
Subroutine Specification

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

Lines: 5000 to 5050

Input parameter: X1 has value of number

Local variables: NN$, PP



The corresponding code is:

```
5000 REM DISPLAY X1 TO 3 DECIMAL
     PLACES
5010 NN$ = STR$(X1 + 0.0005)
5020 FOR PP = 1 TO LEN(NN$)
5030 IF MID$(NN$,PP,1) = "." THEN PRINT
     LEFT$(NN$,PP + 3);:RETURN
5040 NEXT PP
5050 PRINT NN$;".000";:RETURN:REM
     NO DECIMAL POINT IN NN$
```

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

```
10 FOR J = 1 TO 7 STEP 0.1
20 X1 = J: GOSUB 5000
30 X1 = J * J: GOSUB 5000
40 PRINT
50 NEXT J
60 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

```
0.5*10↑-Y1
```

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

Experiment 21.2 Completed	

AVOIDING WORD OVERFLOW ON THE SCREEN

Home computers which use television sets for their screens are generally limited in the number of characters they can display across the screen. This is not the fault of the computer, but arises because a domestic television receiver cannot handle a sufficiently detailed picture to show small characters legibly. The 64 gives you 40 characters in each line. This is as good as any other home computer, and better than most. If you want more than 40 characters, you'll have to buy a much more expensive machine with its own monitor screen.

The screen width on the 64 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 40 characters long.

(2) If a line is exactly 40 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 40th 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 the string, it will be chopped up into 40-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 40 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 41st character and searching *backwards*. To see why, consider the string

FRIENDS, ROMANS, COUNTRYMEN,
LEND ME YOUR EARS

The 41st character is the R in YOUR, so we search backwards until we come to the space at character 37.

We display the 36 character line

FRIENDS, ROMANS, COUNTRYMEN, LEND ME

and remove 37 characters from the front of the string, leaving

This now fits easily on to the next line.

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

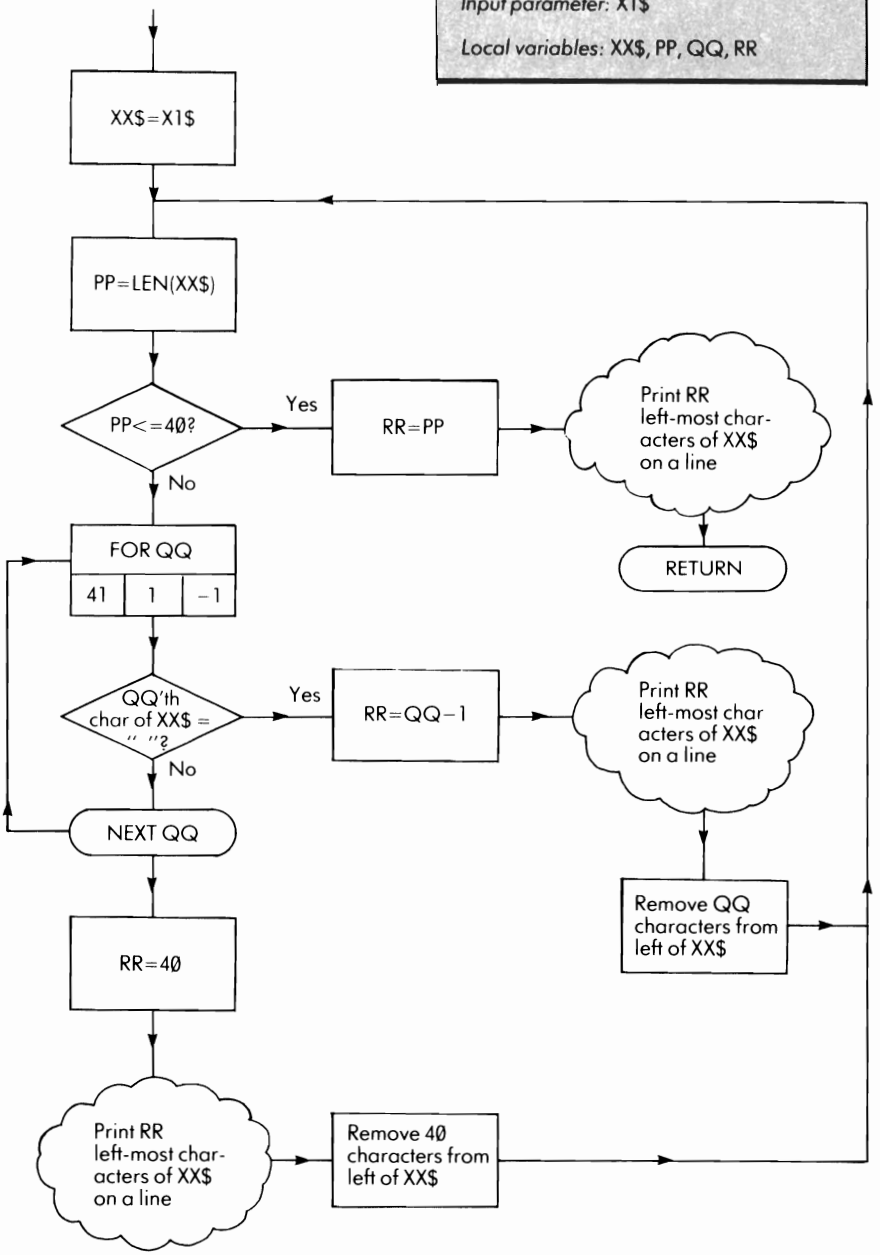
Subroutine Specification

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

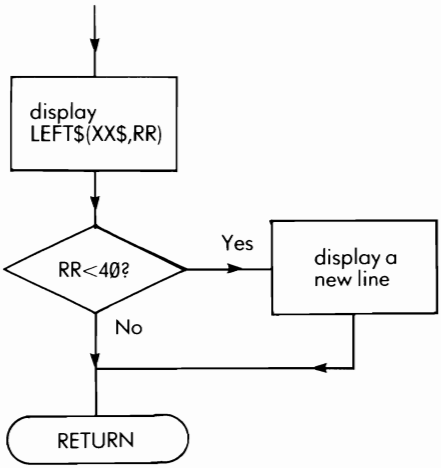
Lines: 5500-5600

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$)
5530 IF PP<=40 THEN RR=PP:GOSUB
     5580:RETURN
5540 FOR QQ=41 TO 1 STEP -1
5550 IF MID$(XX$,QQ,1)=" "THEN
     RR=QQ-1:GOSUB 5580:
     XX$=RIGHT$(XX$,PP-QQ):GOTO
     5520
5560 NEXT QQ
5570 RR=40: GOSUB 5580: XX$=RIGHT$
     (XX$,PP-40): GOTO 5520
5580 REM INTERNAL SUBROUTINE
5590 PRINT LEFT$(XX$,RR);: IF RR<40
     THEN PRINT
5600 RETURN
```

A suitable driver program for this subroutine is

```
10 X1$="TYPE ANY STRING AT ALL UP TO
   TWO LINES LONG"
15 X1$=X1$+" TO TRY OUT THE LAYOUT
   SUBROUTINE"
20 GOSUB 5500
30 INPUT X1$: 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 UNIT21QUIZ64.

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".

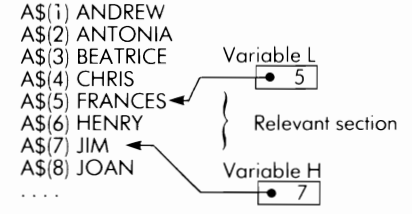
Let's code the method in BASIC. We assume that the list to be searched is 100 items long and can be found in elements A$(1) to A$(100) of array A$. The target word is X$.

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:



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.

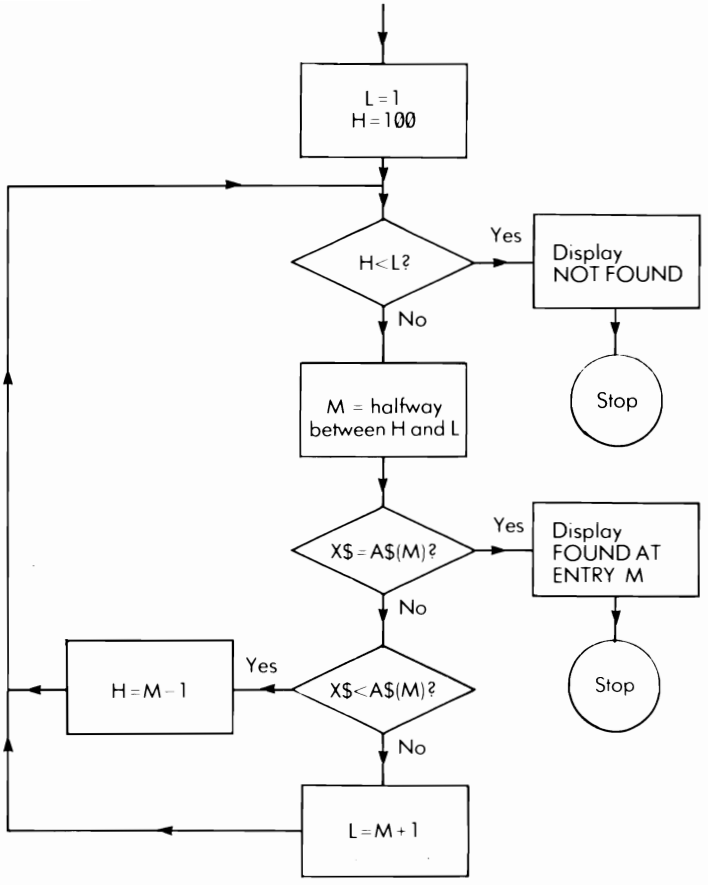
Finding the 'middle' word of the interesting part is quite easy. Its subscript is the 'average' of H and L, reduced to a whole number if necessary. The appropriate expression is

```
INT(0.5*(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$: Target word
A$(1-100): 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: 6000-6100

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:

```
10 DATA BAIN, BEAVIS, BOWEY, BURNS,
   CLARK, FLEMING
20 DATA GORDON, GREEN, HOOD,
   KIDD, MACCABE, MALLY
30 DATA MARSHALL, MILLER, NORTH,
   PACK, PERKINS, REED, ROSE
40 DATA ROSS, SIMPSON, SMITH, SYKES,
   TEDFORD, WEBSTER, WOOD
50 DIM A$(26)
60 FOR J=1 TO 26 : READ A$(J) : NEXT J
70 INPUT "TYPE A NAME"; X$
80 L1=1 : H1=26 : GOSUB 6000
90 IF M1=-1 THEN PRINT X$; " NOT
   FOUND": GOTO 70
100 PRINT X$; " FOUND AT ENTRY";M1
110 GOTO 70
```

Experiment 22.1 Completed

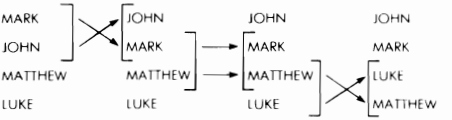
THE BUBBLE SORT

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.

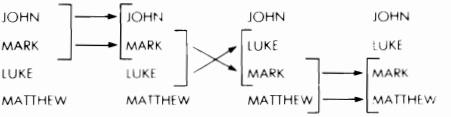
In a sorted list of names you can take any pair, and the one with the larger subscript will be alphabetically greater or equal to the one with the smaller subscript. This fact is the basis of the 'Bubble sort', which is the simplest method of sorting.

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.

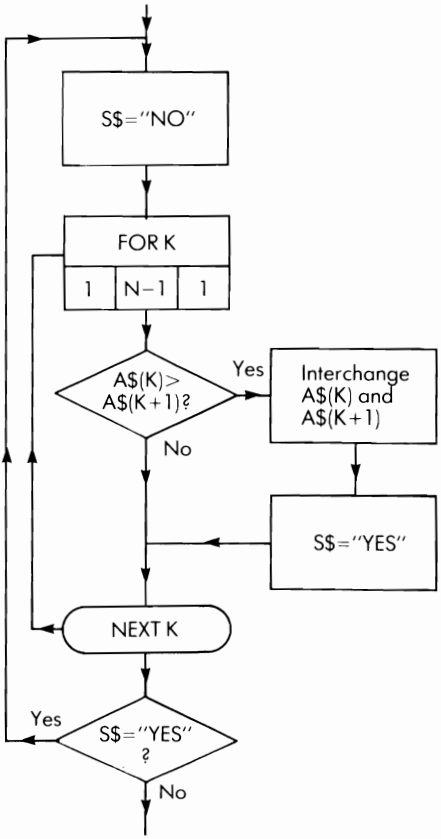
Interchanging two variables is not quite as simple as it seems. If you try to swap the values of X and Y by writing

```
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

```
....
130 S$="NO"
140 FOR K=1 TO N-1
150 IF A$(K)>A$(K+1) THEN D$=A$(K):
    A$(K)=A$(K+1):A$(K+1)=D$:
    S$="YES"
160 NEXT K
170 IF S$ = "YES" THEN 130
....
```

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: 6500-6580

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	

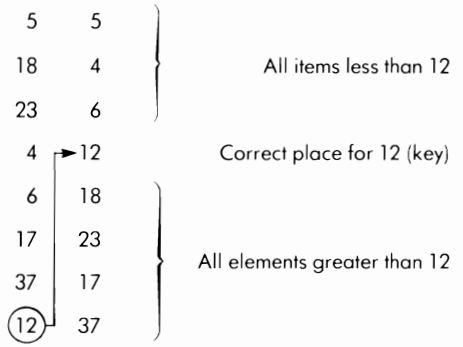
QUICKSORT

The bubble sort is a fine simple method if the list is quite short (say 10 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 50 items will take about 25 times as long to be sorted as a list of 10 items.

There exist 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:



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 6170 is equivalent to a RETURN.

In the subroutine, lines 6040 to 6090 carry out stage 1; 6100 to 6130 are concerned with stage 2 (which can be skipped if the 'list' above the key element is less than two items long). Lines 6140 to 6160 look after stage 3, and lines 6010, 6020, 6110, 6130, 6150 and 6170 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: 6000-6180

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(N1)

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

```
6000 REM QUICKSORT : SORTS N1
     ELEMENTS OF A1
6010 IF SS=1 THEN 6030
6020 DIM SS%(N1) : SS=1 : REM DECLARE
     STACK
6030 AA=1 : BB=N1: SS%(0)=1:PP=1
6040 XX=AA: YY=BB: ZZ=A1(BB)
6050 IF XX>=YY THEN 6090
6060 IF A1(XX)<=ZZ THEN XX=XX+1:
     GOTO 6050
6070 IF A1(YY)>=ZZ THEN YY=YY-1 :
     GOTO 6050
6080 DD=A1(YY): A1(YY)=A1(XX):
     A1(XX)=DD: GOTO 6050
6090 A1(BB)=A1(XX): A1(XX)=ZZ
6100 IF XX-AA<= 1 THEN 6140
6110 SS%(PP)=XX: SS%(PP+1)=BB:
     SS%(PP+2)=2: PP=PP+3
6120 BB=XX-1: GOTO 6040
6130 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 or diskette.

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 yourself 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 64 and timing them. The times are in seconds.

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 COMMODORE 64 MEMORY

When compared to most other home computers, the 64 has a spacious memory capacity. Nevertheless, when you start using arrays, you may eventually come up against the space limitations of the 64 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 some additional overheads for each array.

In this section, we shall take a first look at the way the 64 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 computer on, the message comes up:

64K RAM SYSTEM 38911 BASIC BYTES FREE

(As you will know, the 64 has a total of 65536 bytes of RAM, but a good number of them are allocated to other purposes.)

If you now type

PRINT FRE(0)

the machine replies −26627.

When the FRE(0) function is used on the 64 it will frequently return a negative result, this is because of the way the operating system calculates the result of the function. When you ask for FRE(0) an integer result is calculated. Integers can hold any value between −32768 and +32767. To fully understand why this causes the FRE function to give negative results would involve a considerable detour into binary arithmetic and variable storage in the 64. Suffice to say that a very large number (greater than 32767) becomes negative, to calculate the true "bytes free" you must add 65536 to the result.

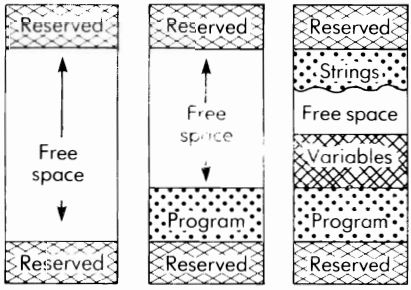
eg. If FRE(0) returns −26627 adding 65536 will give 38909

This is two less than the value on power up, because the FRE(0) function uses two bytes to do its calculations.

A general formula to calculate the true "free bytes" is:

free =(−65536 ★ (FRE(0)<0)) + FRE(0) +2

The overall situation is shown in part (i) of the diagram below. Of the 65536 bytes in the 64, 26625 are reserved for various purposes, 2 bytes are used up in obeying the FRE function itself, but the rest are still free.



(i) (ii) (iii)

Next, you might type in a program or load one from a cassette tape or diskette. 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 64 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 program, 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)
60 PRINT FRE(0), 50
70 DIM N$(5)
80 PRINT FRE(0),70
90 C$="A STRING"
100 PRINT FRE(0), 90
110 D$ = "ANOTHER "+" STRING"
120 PRINT FRE(0), 110
130 D$=""
140 PRINT FRE(0), 130
150 C$=""
160 PRINT FRE (0), 150
170 STOP
```

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:

	0	1	2	3	4	5	6	7
0	X(0,0)	X(0,1)	X(0,2)	X(0,3)	X(0,4)	X(0,5)	X(0,6)	X(0,7)
1	X(1,0)	X(1,1)	X(1,2)	X(1,3)	X(1,4)	X(1,5)	X(1,6)	X(1,7)
2	X(2,0)	X(2,1)	X(2,2)	X(2,3)	X(2,4)	X(2,5)	X(2,6)	X(2,7)
3	X(3,0)	X(3,1)	X(3,2)	X(3,3)	X(3,4)	X(3,5)	X(3,6)	X(3,7)
4	X(4,0)	X(4,1)	X(4,2)	X(4,3)	X(4,4)	X(4,5)	X(4,6)	X(4,7)
5	X(5,0)	X(5,1)	X(5,2)	X(5,3)	X(5,4)	X(5,5)	X(5,6)	X(5,7)

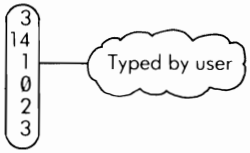
Each element in the array has *two* subscripts: a row number and column number. For example, X(3,4) is in row 3 and column 4. Apart from this fundamental difference, everything you know about one-dimensional arrays can be extended to two dimensions.

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:

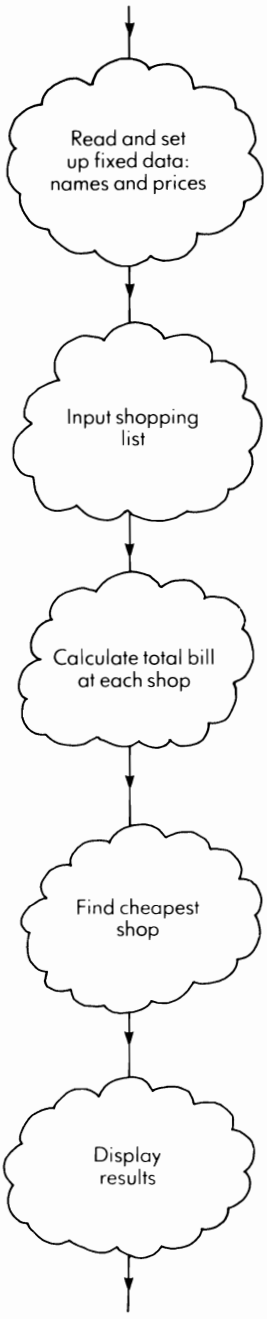
	FINE FARE	ASDA	SAINS-BURYS	CO-OP	FRASERS
FLOUR	29	31	27	26	32
POTATOES	15	12	13	24	33
BUTTER	47	49	40	45	39
SUGAR	22	20	19	27	29
CHEESE	94	80	103	107	99
APPLES	32	18	22	27	21
	(All prices in pence per pound)				

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
SUGAR? 0
CHEESE? 2
APPLES? 3

BEST VISIT ASDA
WHERE YOU PAY
3 LBS FLOUR       :  93
14 LBS POTATOES:  168
1 LB BUTTER       :  49
0 LBS SUGAR       :   0
2 LBS CHEESE      : 160
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 0. This is common in small problems.

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

```
 10 DATA FLOUR, POTATOES, BUTTER,
    SUGAR, CHEESE, APPLES
 20 DATA FINE FARE, ASDA, SAINSBURYS,
    COOP, FRASERS
 30 DATA 29, 31, 27, 26, 32
 40 DATA 15, 12, 13, 24, 33
 50 DATA 47, 49, 40, 45, 39
 60 DATA 22, 20, 19, 27, 29
 70 DATA 94, 80, 103, 107, 99
 80 DATA 32, 18, 22, 27, 21
 90 DIM F$(6), S$(5), F(6), T(5), P(6,5)
100 FOR K=1 TO 6 : READ F$(K) : NEXT K
110 FOR J=1 TO 5: READ S$(J): NEXT J
120 FOR K=1 TO 6: FOR J=1 TO 5
130 READ P(K,J)
140 NEXT J,K
150 PRINT " SHIFT and CLR HOME PLEASE
    STATE YOUR"
160 PRINT "NEEDS IN LBS"
170 FOR K=1 TO 6
180 PRINT F$(K);: INPUT F(K)
190 NEXT K
200 FOR J=1 TO 5
210 FOR K=1 TO 6
220 T(J)=T(J)+F(K)*P(K,J)
230 NEXT K,J
240 M=T(1): N=1
250 FOR J=2 TO 5
260 IF T(J)<M THEN M=T(J): N=J
270 NEXT J
280 PRINT "BEST VISIT "; S$(N)
290 PRINT "WHERE YOU'LL PAY" : PRINT
300 FOR K=1 TO 6
310 PRINT F(K); "LB . ";
320 IF F(K)<>1 THEN PRINT " SHIFT
    and CRSR S. ";
```

```
325 X = F(K) * P (K,N)
330 PRINT F$(K); TAB(15);

340 PRINT LEFT $ ( RIGHT $ ( "          SPACE
    11 times " + STR$ (X),10),10)
345 NEXT K
350 PRINT: PRINT " TOTAL = " ; TAB (15);:

    PRINT LEFT $ ( RIGHT $ ( "          SPACE
    11 times " + STR $ (M), 10),10)
360 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 80 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 350 makes the machine move its internal cursor to the 15th column in the screen (if not already past it). 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

```
10 FOR J=20 TO 1 STEP -1
20 PRINT TAB(J); "/"
30 NEXT J
40 STOP
```

will display a sloping line across the screen.

Note lines 340 and 350 give right justification for the price values.

In the program on the previous page, commands 310-320 are purely concerned with displaying "LB . " for one pound, or "LBS . " if more (or less) than one. 310 displays "LB . ", and 320 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 has 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 100 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 diskette or cassette tape, and find the 'minimum' pass-mark a quarter way down the sorted list. Use it to pick out the students who pass.

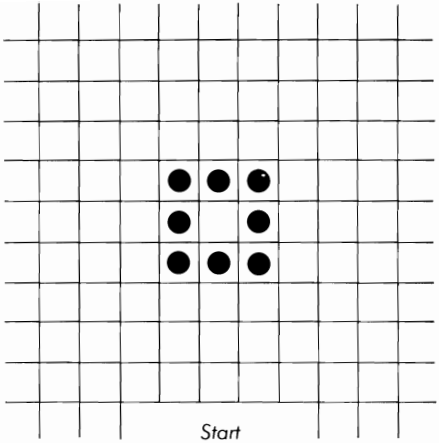
(b) The game of 'life' was invented by R. Conway, an English Mathematician. It concerns the life history of a colony of bugs which live in a rectangular area, one to each cell.

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

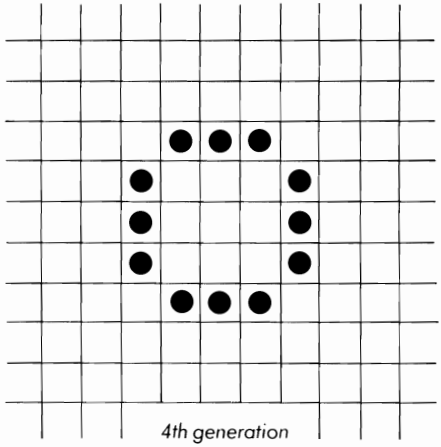
1. If a bug has 1 or fewer immediate neighbours, it dies of loneliness. Cells with touching corners are also counted as neighbours.
2. If it has 4 or more neighbours, it dies of overcrowding.
3. If it has 2 or 3 neighbours it survives to the next generation.

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

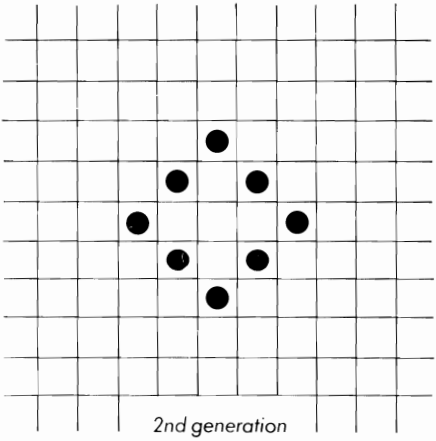
To give an example, consider



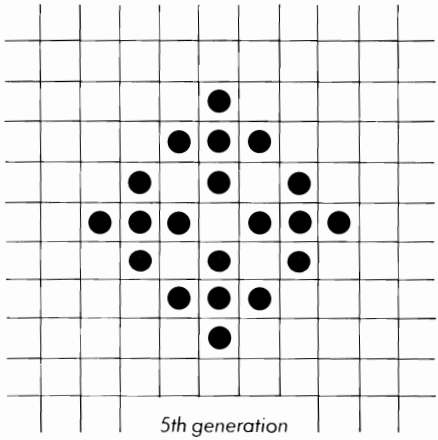
Start



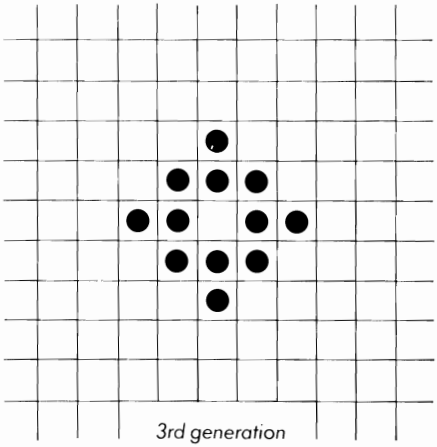
4th generation



2nd generation



5th generation



3rd 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 or diskette. The program reads a starting position into array X$(9,9) (lines 10-180) and then displays it (lines 190-260). The partial program also declares and array Y$(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 COMMODORE 64

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, some people will always think of the COMMODORE 64 just as a machine which plays games which come on cassette or plug-in cartridges. If you don't want to know about programming, this is a perfectly reasonable and useful level of understanding.

Some people like to delve deeper. You, the reader, are already aware of the 64 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 64. You'll find that the description of the 64'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 many sections of the book, the material in this Unit applies only to the 64 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 64 than other BASIC commands. This means that a level of protection is by-passed. A program with errors can corrupt the computer'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 destroys your program, it is doubly important frequently to store the program on a cassette or a diskette, before you run it, if you are using many PEEKs and POKEs.

Software corruption is a temporary effect. It is absolutely impossible to do your 64 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 more about the 64 computer itself.

What the book says is quite complicated, so don't be surprised if you have to read it over several times before you really understand all the finer points.

BITS, BYTES AND ADDRESSES

The fundamental unit of information inside a computer is the binary digit or *bit*. A bit can have only two possible values: 0 and 1.

Bits by themselves are not particularly useful. In the COMMODORE 64 (and indeed in most other computers) they are collected together in groups of eight, called *bytes*. There are 256 possible combinations of bits in a byte. Some of them are:

00000000 01011100 10101011 11111111

The *meaning* of a byte depends entirely on the context in which it is used. In the 64 it could be

(a) A code for a specific character (e.g., "A" could be 00000001).

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

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

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

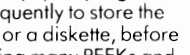
(e) One of a group of *five* bytes which together make up the value of a number variable.

(f) One of several bytes representing the characters in a string variable.

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

The leftmost bit scores 128 if 1, zero if it is 0
The next bit scores 64 if 1, zero if it is 0
The next bit scores 32 if 1, zero if it is 0
The next bit scores 16 if 1, zero if it is 0
The next bit scores 8 if 1, zero if it is 0
The next bit scores 4 if 1, zero if it is 0
The next bit scores 2 if 1, zero if it is 0
The rightmost bit scores 1 if 1, zero if it is 0.

To give an example, consider the byte



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

The scores for the bits are easy to remember because they start at 128, and then each one is exactly half the one which came before.

Sometimes you have to convert a number into its binary 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 Ø.

2. If you can subtract 64 from the number left after step 1, then do so and write down a 1 to the right of the previous symbol. Otherwise write down a Ø to the right of the previous symbol.

3. to 8.: Do the same for 32, 16, 8, 4, 2 and 1. You will now have a row of 8 bits which is the binary equivalent of the number you started with.

As an example, take the number 201.

(1)	201−128 = 73, so put down	1
(2)	73−64 = 9, so put down	11
(3)	9−32 won't go, so put down	110
(4)	9−16 won't go, so put down	1100
(5)	9−8 = 1, so put down	11001
(6)	1−4 won't go, so put down	110010
(7)	1−2 won't go, so put down	1100100
(8)	1−1 = Ø, so put down	11001001

So the binary form of "201" is "11001001"

Sometimes the computer needs to handle "addresses". An address is a number made up of two bytes side by side. The address with the highest possible value consists of 16 1's: 1111111111111111. If we extend the conversion rules and apply them, we find that the equivalent number is 65535. Since the lowest possible address is zero, it follows that there are 65536 possible different addresses.

THE STRUCTURE OF THE 64

The 64 consists of several units connected together. They are:

(a) A *memory* which contains various types of store:

- *RAM* or Random Access Memory, where the contents can change from moment to moment. RAM is normally used to store BASIC programs and variables.
- *ROM* or Read Only Memory, which contains information built in during the manufacturing process. The contents can never change. ROM is used to store programs (like the Screen Editor) which must always be present in the computer.
- *Peripheral Control Registers* (or "PCRs"), which are parts of the sound and vision control chips, and interface chips used to connect the keyboard, cassette unit, disk drive and printer.

All memory is arranged in groups of bytes. In most cases, the amount of memory in any one category is an exact multiple of 1024 bytes. This quantity of store is often called a Kilobyte, or a "K" for short. Thus the store contains 65536 bytes of RAM ("64K"), 20480 bytes of ROM ("20K"), and space for up to 4096 bytes ("4K") of peripheral control registers.

(b) A *Microprocessor* which takes instructions out of the memory (ROM or RAM) and obeys them. Most instructions result in the contents of the RAM or the peripheral control registers being altered in some way. The individual instructions are roughly similar to the commands in a BASIC program, but they are simpler. They are also executed very much faster — up to half a million every second.

(c) A *Keyboard*, which is looked after by the Microprocessor.

(d) The *Video Controller*, which produces the picture on the TV screen.

(e) The *Sound Controller*, which synthesizes the sounds you hear on the TV set.

(f) Your computer will also have either a *Cassette Unit* or a *Disk Drive*. These units are controlled by the microprocessor and PCRs.

Figure 23.1 is a diagram of the overall structure of the COMMODORE 64.

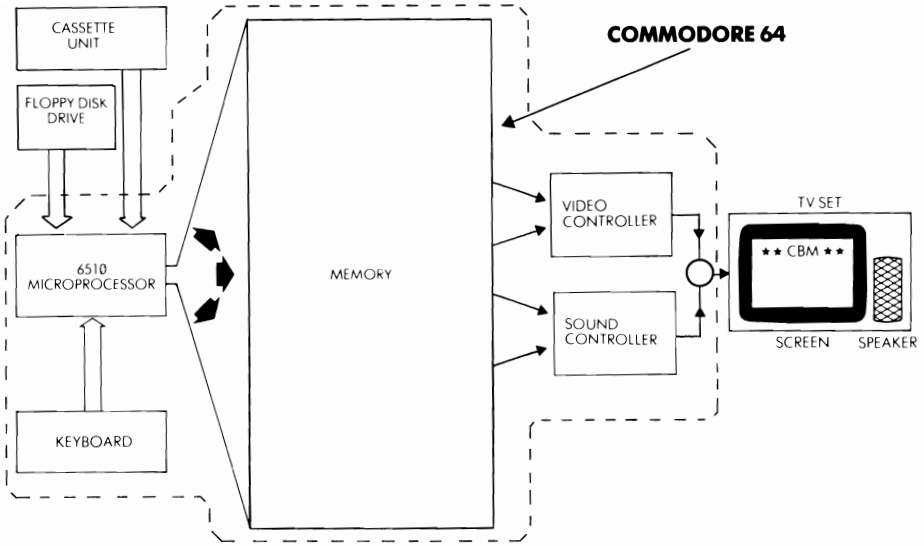


Figure 23.1

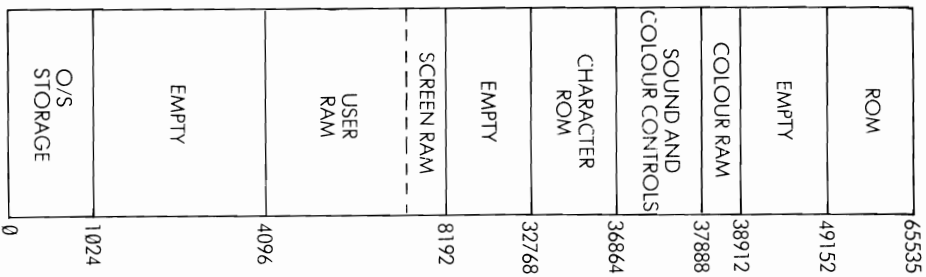


Figure 23.2
Address Map for the VIC 20 (a machine with a one-layer store)

THE COMMODORE 64 MEMORY

The structure of the COMMODORE 64 is quite complex, because the designers have managed to cram more storage into the machine than seems possible at first sight.

In most computers every byte in the memory is given its own unique address. There is a byte with address 0, another one with address 1, and so on. The bytes are put in groups with adjacent addresses, and there are quite often gaps in the address sequence. The COMMODORE VIC 20 illustrates this point perfectly. We can draw an 'Address Map' for the VIC 20 which looks like Figure 23.2.

The map tells us there is RAM between 0 and 1023, and again between 4096 and 8191. ROM, colour RAM and peripheral controllers fill other parts of the address space. There are various empty regions, which can be filled by cartridges and these plug into the extension slot at the back of the computer.

There are clear advantages in such a scheme, where each byte in the store has its own special address. When the microprocessor wants to fetch or alter the contents of a particular cell, all it has to do is quote the correct address. This seems so obvious it is hardly worth saying, but read on!

Unfortunately, this scheme won't work on the 64, because — as you may have guessed — the total amount of store is considerably more than the number of different addresses you can make with 2 bytes or 16 bits.

The way round the problem is to use a layered store. The lowest layer is the 64K of RAM, but parts of it are concealed by other types of memory. The diagram below (Figure 23.3) shows two views of the 64 memory. When the computer is running ordinary BASIC programs, it 'sees' only the top view, which includes about 40K of RAM, 20K of ROM and 4K of peripheral control registers.

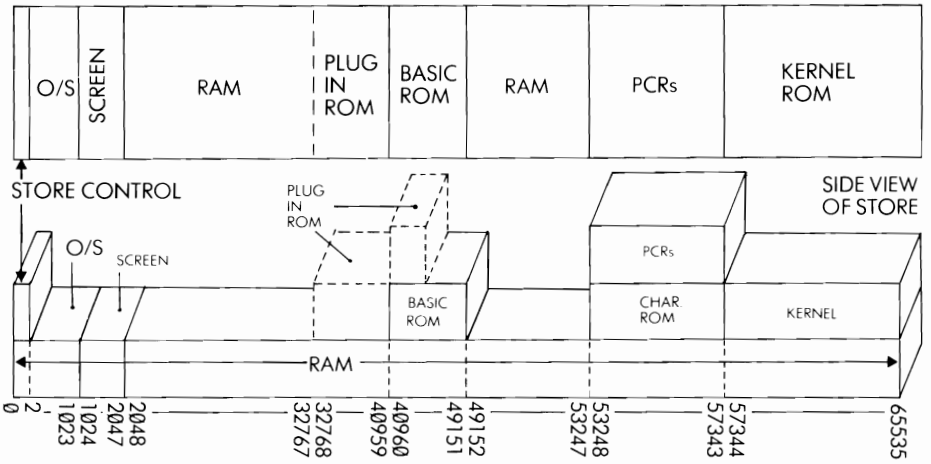


Figure 23.3
Address Map for COMMODORE 64

How does the microprocessor gain access to the hidden part of the store? The key is in the register at address 1, which acts something like a Peripheral Control Register. Each bit in the register controls one of the blocks of memory which hide other blocks. Such a block can be switched out of view, revealing a different type of store below. For instance, the right-most bit in register 1 controls the ROM with the BASIC interpreter. If you change this bit from 1 to zero, then the interpreter will disappear and the microprocessor will have direct access to the underlying 8K of RAM. Of course you can't run BASIC programs when the machine is in this condition, but there is more RAM available for programs written in other programming languages like PASCAL or FORTH.

In a similar way, the second right-most bit removes the KERNEL software, which normally looks after peripheral devices. The third right-most bit lets the microprocessor see the Character ROM, which is normally concealed by the Peripheral Control Registers.

There is no way of removing the Store Control Registers themselves. They are always available at addresses 0 and 1, and bytes 0 and 1 of the RAM are permanently obscured by them.

Unless you are a professional programmer, you are unlikely to have much occasion to set aside the BASIC and KERNEL ROMs. On the other hand, it is often useful to get access to the Character ROM, and we shall be showing you how and why to do it later in this unit.

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. If the address is in part of the store where one type of memory can hide another, the byte chosen is the one visible to the computer at that moment.

You can apply the PEEK function to any of the 65536 possible addresses. If you choose a visible area with ROM (such as "60000") you can find out what pattern was put into that byte when the computer was manufactured.

To get another view of the way bytes can be used, let's use PEEK to have a look at the Character ROM. This section of memory contains the *shapes* of the various characters as they appear on the screen.

The Character ROM is normally concealed by the Peripheral Control Registers, so a sensible way to examine it at leisure is to copy its contents into a section of RAM which is always visible to the computer. Enter and run the following program, making quite sure that you've typed it correctly before giving the RUN command:

```
10 A= 14336 : B = 53248 : C = 56334
20 POKE C, PEEK(C) AND 254
30 POKE 1, PEEK(1) AND 251
40 FOR J = 0 TO 2047:POKE A+J,PEEK
   (B+J): NEXT J
50 POKE 1, PEEK(1) OR 4
60 POKE C, PEEK(C) OR 1
70 STOP
```

This program will copy the first 2048 bytes of the Character ROM and put them into RAM addresses 14336 onwards, taking about 15 seconds to do so.

The details of the program may be a little obscure to you. When the Peripheral Control Registers are by-passed (as they must be for this

program to work), they must first be switched off or 'disabled', otherwise they will interfere with the program and make the machine stop. Line 20 in the program has the special task of disabling the PCRs. Line 30 alters the value of register 1, so that the Character ROM comes into view for the microprocessor. Line 40 moves the bytes, and lines 50 and 60 restore the original state, so that ordinary programs can run again.

When you've run the program, you can use PEEK to inspect the contents of bytes 14336 onwards, which now contain an exact (but "visible") copy of part of the Character ROM. Start with the following command in immediate mode (all on one line):


```
FOR J=14344 TO 14351:PRINTJ;PEEK(J):
NEXT J
```

You get:

```
14344  24
14345  60
14346  102
14347  126
14348  102
14349  102
14350  102
14351  0
```

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

```
 24 : 00011000          11
 60 : 00111100         1111
102 : 01100110       11  11
126 : 01111110       111111
102 : 01100110       11  11
102 : 01100110       11  11
102 : 01100110       11  11
  0 : 00000000
```

The letter "A" pattern is now clear. When the 64 displays an "A" on the screen it uses these 8 bytes from the ROM to get the right shape of the letter.

EXPERIMENT 23·1

(a) Using PEEKs, find out what character is stored in addresses 14416 to 14423. Then look at addresses 14496 to 14503. Does this help you guess a relationship between the characters and their positions in the ROM?

(b) Write a program which uses the subroutines below to expose the Character ROM and display magnified versions of some of the characters in it.

```
1000 REM GIVEN A BYTE IN X1, WORK
     OUT THE CORRESPONDING BINARY
     PATTERN
1010 REM AND DISPLAY IT AS A ROW OF
     STARS AND SPACES
1020 YY = 256: FOR KK = 1 TO 8
1030 YY = YY/2
1040 IF X1>=YY THEN X1 = X1 - YY:
     PRINT"★";:GOTO 1060
1050 PRINT " ";
1060 NEXT KK
1070 PRINT: RETURN

1500 REM GIVEN AN ADDRESS IN Y1,
     RETURN THE CONTENTS OF THAT
     ADDRESS
1510 REM IN THE CHARACTER ROM IN X1
1520 C = 56334
1530 POKE C, PEEK(C) AND 254: POKE 1,
     PEEK(1) AND 251
1540 X1 = PEEK(Y1)
1550 POKE 1,PEEK(1) OR 4: POKE C,
     PEEK(C) OR 1
1560 RETURN
```

Experiment 23.1 Completed	

THE POKE COMMAND

POKE is by now an old friend. This section tells you a few things you may not have known.

The POKE command uses two arguments: an address (in the range 0 to 65535) and a number between 0 and 255. The effect is to store the binary pattern of that number in the selected address. For example,

```
POKE 54296 ,15
```

puts the binary pattern 00001111 into location 54296. Of course the command only works correctly if the selected address is RAM or a Peripheral Control Register. If you POKE into a ROM address (such as "6000") the byte goes into the hidden RAM beneath. You cannot get it out again unless you remove the ROM by changing a bit in the Store Control Register. A POKE may have weird effects if you choose a RAM address which belongs to the KERNEL. To illustrate the point, try POKE 788,44 ; you'll have to switch your machine off and on again to regain control.

One important use for the POKE command is to drive the Video Controller. The controller is connected to *four* different areas of store:

(a) The special Peripheral Control Registers (such as 53280 and 53281, which control the frame and background colours).

(b) The screen RAM, which is an area of RAM between 1024 and 2047. Each byte (except the last 24) holds a code for a character to be displayed on the screen.

(c) The colour RAM, which lies between 55296 and 56295. Each byte indicates the *colour* of a character to be displayed.

(d) The Character ROM, which — as we have seen — contains the *shapes* of the various characters.

The picture you see on the screen is drawn afresh 50 times every second. Each time round, the generator looks at locations 53280 and 53281 and paints the frame and background colours accordingly. Then it paints each of the 1000 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 draw each character is quite complex. To paint the first character, here's what happens:

The generator begins by looking in address 1024, which is the first location in the screen RAM. Here it finds a *screen code* which tells it what character is needed. The code is shown in Figure 23.4, where you should ignore the column marked "SET 2" for the present. Using this code sheet, you will see that an 'M' is represented by 13, or a ♥ by 83.

A screen code can have any value between 0 and 255, but the code 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 Video Controller looks in the Character ROM to find out what shape to paint. To find the right *shape* it multiplies the screen code by 8, adds on 53248 and fetches 8 bytes starting with the calculated address. For instance, it gets the shape of the 'M' from the 8 bytes starting at 53248 + 13★8 or 53352. The Character ROM is always visible to the Video Controller, even though it may be hidden from the 6510 microprocessor.

Lastly, the controller goes to the colour RAM and fetches the first byte, at 55296. This tells it the *colour* of the first character according to the standard code:

0	1	2	3	4	5	6	7
Black	White	Red	Cyan	Purple	Green	Blue	Yellow

8	9	10	11	12	13	14	15
Orange	Brown	Light Red	Grey 1	Grey 2	Light Green	Light Blue	Grey 3

The Video Controller now has enough information to draw the first character in 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 looks in address 1025 to get the screen code, and in 55297 for the colour.

The Video Controller continues this way until it has painted all 40 characters in the first row, then it paints the second row, the third row, and so on until the whole screen is full. Then it starts all over again.

This system seems complicated, but it is very flexible. The controller runs all the time and works quite independently of the microprocessor. To display any information, all that is necessary is to record the right codes in the Screen and Colour RAMs. As soon as this is done the new character appears on the screen within 1/50 of a second.

Suppose you want to display a white diamond 6 lines down and 14 columns across. Each whole line accounts for 40 characters, so this position is displayed 6★40+13 or 253 characters after the top left-hand one. The corresponding cell in the screen RAM is 1024 + 253 = 1277, and in the colour RAM it is 55296 + 253 = 55549.

According to Figure 23.4, the code for a diamond is 90, and "white" is 1, so the following pair of commands should put a white diamond in the right place:

```
POKE 1277,90: POKE 55549,1
```

SCREEN CODES

SET 1	SET 2	POKE
@		0
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	10
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	20
U	u	21
V	v	22
W	w	23
X	x	24
Y	y	25
Z	z	26
[		27
£		28
]		29
↑		30
←		31
SPACE		32
!		33
"		34
#		35
$		36
%		37
&		38
'		39
(		40
)		41
★		42
+		43
,		44
—		45
.		46
/		47
0		48
1		49
2		50
3		51
4		52
5		53
6		54
7		55
8		56
9		57
:		58
;		59
<		60
=		61
>		62

Figure 23.4

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

SET 1	SET 2	POKE
	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

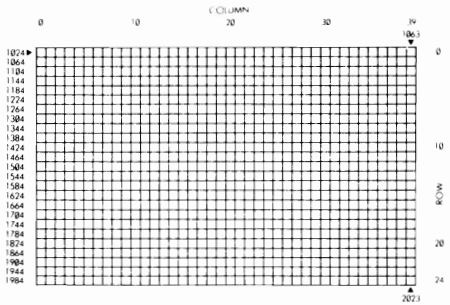
SET 1	SET 2	POKE
		106
		107
		108
		109
		110
		111
		112
		113
		114
		115
		116
		117
		118
		119
		120
		121
	√	122
		123
		124
		125
		126
		127

Figure 23.4 continued

The diagrams in Figure 23.5 will help you work out the right RAM addresses 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 purple diagonal line up the screen:

```
10 PRINT " SHIFT and CLR HOME "
20 POKE 53281,1 : REM PAINT SCREEN
   WHITE
30 FOR J = 32 TO 968 STEP 39
40 POKE 1024 + J, 78: POKE 55296+J,4
50 NEXT J
60 GOTO 60 : REM LOOP STOP
```


SCREEN MEMORY MAP



COLOUR MEMORY MAP

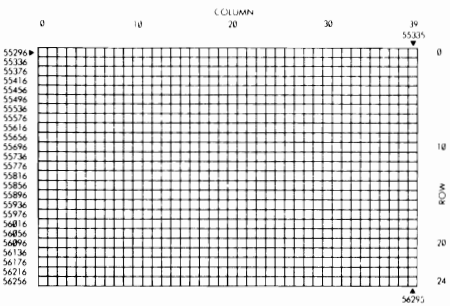


Figure 23.5

EXPERIMENT 23·2

Here is a slightly longer program, MONDRIAN, which uses POKEs to create an artistic effect. Try it out and then do your best to improve it.

```
5 REM MONDRIAN (WITH APOLOGIES)
10 PRINT " SHIFT and CLR HOME "
20 FOR L=1 to 8
30 FOR K=0 to 7
40 POKE 53281, K+L
50 FOR J=K TO 9
60 Y1=2+J
70 FOR X1=2+J TO 20-J
80 GOSUB 1000
90 NEXT X1
100 X1=20-J
110 FOR Y1=2+J TO 20-J
120 GOSUB 1000
130 NEXT Y1
140 Y1=20-J
150 FOR X1=20-J TO 2+J STEP-1
160 GOSUB 1000
170 NEXT X1
180 X1=2+J
190 FOR Y1=20-J TO 2+J STEP-1
200 GOSUB 1000
210 NEXT Y1
220 NEXT J,K,L
230 GOTO 230: REM LOOP STOP
1000 REM DRAW REVERSED SPACE AT
     X1+6,Y1+1. USE COLOUR K
1010 ZZ=X1+40*Y1+46
1020 POKE 1024+ZZ, 160: REM 160 IS
     SCREEN CODE FOR REVERSED SPACE
1030 POKE 55296 + ZZ , K
1040 RETURN
```

Experiment 23.2 Completed	

THE CHARACTERS IN SET 2

If you look at Figure 23.4 again, you will see that the columns marked SET 2 define a different set of characters. Some of the characters (those where the SET 2 column is empty) are the same as SET 1, but the majority are changed. In particular, all the screen codes which give you capitals in SET 1 correspond to lower case letters in SET 2, whilst some of the graphics in SET 1 are now replaced by the capital letters. SET 2 is particularly suitable for programs which handle text, such as word processors.

There are two ways to select SET 2.

1. When you are editing a program, hold down SHIFT and hit the C= key. Do it again and the machine goes back to SET 1.

2. When a program requires to use SET 2, it should give the command

   POKE 53272,23

   To go back to SET 1, the command is

   POKE 53272,21

Notice that it is difficult to use characters from both sets at the same time, unless you construct your own character definitions. This is explained in the next section.

DEFINING YOUR OWN CHARACTERS

You've seen how the COMMODORE 64 decides the shapes of the various characters it displays on the screen, and you know how to copy the bytes which describe the characters into RAM where they are accessible. The Video Controller chip is so flexible that it can be told to look in the RAM for its character shapes instead of the Character ROM. From here it's only a short step to designing and using your own characters instead of those supplied ready-made in the machine.

The ability to use any character set you like adds another layer of flexibility to the 64. For example, you can now display many different languages such as Russian, Greek, Arabic, Hebrew or Malayalam. You can also define characters to represent monsters, space-ships, and other interesting objects for use in games and educational programs.

To begin, you must design the characters you are going to use. The process is easy but quite laborious. A method you may find useful is this:

Draw a square about 2 inches high, and sketch your character inside it. Keep away from the edges unless you want the character to touch the adjoining characters. All the strokes should be thick and bold.

Next, fill in the square with an 8 by 8 grid. Identify the cells which are entirely or mainly inside the black parts of the character, and mark them with X's. Try to ensure that all your vertical and diagonal strokes are at least 2 cells wide.

Thirdly, copy down each row as a binary number, putting 1's for the cells with X's and 0 for the empty cells.

Finally, convert each of these numbers into its decimal equivalent.

Figure 23.6 shows the process in full for the Russian letter Я ("yat' ").

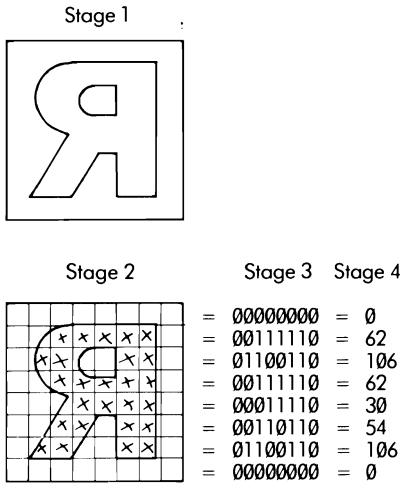


Figure 23.6

When you've designed the new characters, you can begin work on the program which is to use them. In most cases, you will want to use many of the ordinary characters as well as the ones you have defined. The description which follows assumes that your own 'private' character set is based on one of the ordinary ones (SET 1 or SET 2) with just a few changes.

Begin by studying the character sets in Figure 23.4. Select the set you like best and then decide which characters in it you are going to replace by the ones you have designed yourself. People usually choose to discard some of the more obscure graphics symbols, or perhaps the reversed letters. (Remember that the 128 'ordinary' symbols are followed by the 128 reversed ones, in the same order.)

To get the new characters working, your program will have to go through the following steps:

1. Copy one of the standard sets of characters from the Character ROM into locations 14336 onwards. Remember that each set is 2048 bytes long, and that SET 1 starts at 53248, whilst SET 2 starts 2048 bytes higher at 55296.

2. *Replace* the definitions of the standard characters you *don't* need by the descriptions of your own new characters. To do this, your program will need to know

   a) the address of the character to be replaced, and

   b) the shape of the new character.

The address of a character in the RAM is simply 14336 plus eight times the screen code. For instance, the address of the ★ in SET 1 is 14336 + 8★42 or 14672. The address of a reversed character is 1024 bytes higher than that of its 'normal' equivalent.

The shape of the new character is given by the list of 8 numbers you worked out during the last stage of the design process.

It is convenient to collect all the information about a new character into a single DATA statement, with 9 numbers: first the address of the character to be replaced, and then the 8 numbers which define the new shape. For example, we could decide that the definition of the Russian letter Я will replace the digit 1. Since the screen code of 1 is '49', and 14336 + 8★49 = 14728, the appropriate DATA statement will read:

```
DATA 14728,0,62,106,62,30,54,106,0
```

and a segment of program to set up the new definition could be:

```
READ X : REM READ ADDRESS
FOR J = 0 TO 7
READ Y : REM READ A BYTE OF
         DESCRIPTION
POKE X + J,Y
NEXT J
```

When all the character definitions have been changed, the final step is to instruct the Video Controller to look in the RAM for the character shapes instead of the ROM. The command to do this is

```
POKE 53272,31
```

Once the definition of a character has been changed, *all* references to that character will produce the new shape. To illustrate this point, load and run program TONGUE, which redefines the digits 1,2,3 and 4 as the Russian characters Я, З, Ы and К and then displays the Russian word ЯЗЫК, which means 'TONGUE' or 'LANGUAGE'.

Note that this example goes against our advice to replace only obscure 'unwanted characters'. There is a good reason!

Try LISTing the program, without resetting the computer. You'll see that all the 1s, 2s, 3s and 4s now appear as Russian letters! A RUN STOP and RESTORE will bring the ordinary character set back.

This program contains all you need to redefine and use your own characters. A listing is given below for you to study.

```
10 REM DEFINING YOUR OWN
   CHARACTERS
20 A=14336:B=53248:C=56334
30 POKE C,PEEK(C) AND 254: REM
   TRANSFER STANDARD SET OF
   CHARACTERS TO 14336
40 POKE1,PEEK(1) AND 251
50 FORJ=0 TO 2047:POKE(A+J),
   PEEK(B+J):NEXTJ
60 POKE 1,PEEK(1) OR 4
70 POKE C, PEEK(C) OR 1
80 REM
90 REM SET UP YOUR OWN CHARACTER
   DEFINITIONS
100 REM EACH LINES HOLDS ADDRESS
    OF CHARACTER FOLLOWED BY 8
    VALUES WHICH DEFINE
110 REM ITS SHAPE. CHARACTERS ARE
    RUSSIAN LETTERS YA, Z, Y AND K.
120 REM AND THEY REPLACE DIGITS 1 2 3
    AND 4
130 DATA 14728,0,62,106,62,30,54,106,0
140 DATA 14736,0,120,6,12,12,6,120,0
150 DATA 14744,0,199,199,243,223,223,
    243,0
160 DATA 14752,0,102,108,120,120,108,
    102,0
200 FOR J=1 TO 4
210 READ X
220 FOR K=0 TO 7
230 READ Y
240 POKE X+K,Y
250 NEXT K,J
260 REM NEXT SWITCH CHARACTER
    DESCRIPTION ADDRESS TO 14336
270 POKE 53272,31
280 REM NOW TRY IT OUT!
290 PRINT" CTRL and 2 SHIFT
    and CLR HOME CRSR↑↓ 5 times CRSR←→ 5 times
    1 2 3 4"
500 STOP
```

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 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

```
10 INPUT " SHIFT and CLR HOME "; X : REM X
   IS A DUMMY VARIABLE
20 S=0
30 FOR J=0 TO 999 : REM SCAN SCREEN
40 IF PEEK(1024+J)<>32 THEN S=S+1 :
   REM 32 IS SCREEN CODE FOR SPACE
50 NEXT J
60 PRINT "NUMBER OF SYMBOLS ="; S
70 STOP
```

There are certain other locations which can be conveniently POKE'd to change the behaviour of the 64 in predictable and useful ways.

For example there is a group of locations which control the behaviour of the keyboard.

★ Repeating keys: When the 64 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 650, which is interpreted as follows:

```
  0 : Only the 'standard' keys repeat
127 : No keys repeat
128 : All kets repeat
```

If you give the command

```
POKE 650, 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 0 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 **'WASPS64', 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 simulated 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

XX$: The number of 'lives' the hunter has left, as a row of three ♥ s. This row is shortened by one every time the hunter is stung, and when no lives are left the game stops.

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°₀. Thus the column position of the first wasp is in W°₀(1,1), and its row position is W°₀(1,2). The second wasp occupies W°₀(2,1) and W°₀(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
7	9
4	17

← This one killed

gives

NA=5

W%

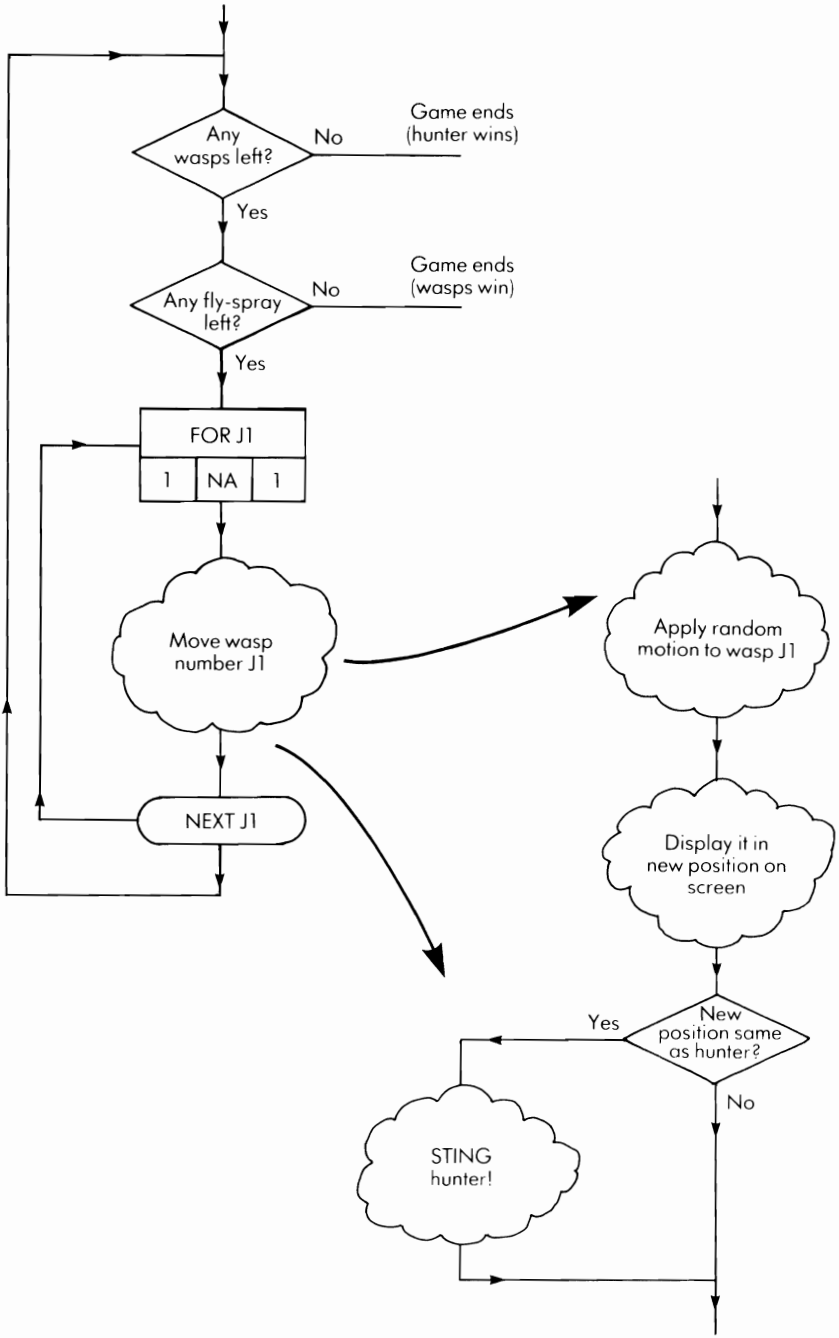
1	17
3	12
18	2
7	9
4	17
—	—

This row unused.

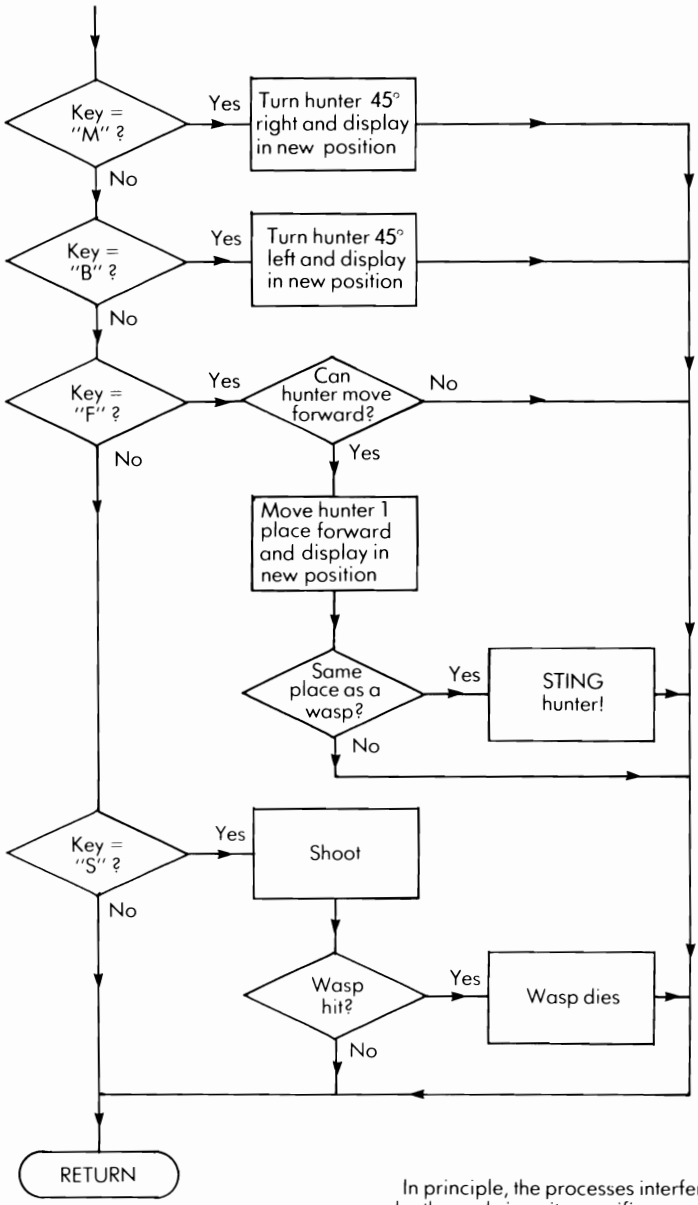
The game itself is organised as two 'processes', which run almost independently: the 'wasp' process and the 'hunter' process.

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 fly 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.

Line 10 declares four arrays. They are all integer arrays, since this will save some space in the store. 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, as we shall explain below.

Lines 20 to 90 display a set of user instructions.

Lines 100 to 118 set up certain numbers in symbolic form. This will make the program easier to read later on. For example, PA is the starting address of the sound generator, and an address like "PA+19" is more comprehensible, and more likely to be correct, than "54291".

These lines also initialise the sound generator, select black as the background colour and make all the keys on the keyboard into 'repeaters'.

Lines 120-130 determine the number of wasps to start with.

Lines 150 to 210 set up appropriate values for the arrays V%, U% and D%.

You'll remember that the squares in the screen RAM are numbered from left to right, and from the top down.

		1256		
	1295	1296	1297	1298
1334	1335	1336	1337	1338
1374	1375	1376	1377	
		1416		

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 40 *less* than the one he occupies himself. Likewise, if he points North-East, the 'aim' cell will be numbered 39 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 U% 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 2000-2020, remembering that the hunter's position is recorded in variables A and B. SR is the constant 1024, 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	0	−1
NE	1	−1
E	1	0
SE	1	1
S	0	1
SW	−1	1
W	−1	0
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.

Line 220 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.

Lines 250 to 290 work out initial positions for all the wasps, and displays them on the screen. At first, all the wasps are put in the top half of the screen (lines 1 to 12). Note (280) the second POKE to set the wasp's colour.

Line 320 fixes the starting position of the hunter and displays him in this position.

Lines 330 to 420 form the heart of the simulation; they run the wasp process and call the hunter process whenever a key is pressed. 340 and 350 display the current state of affairs. Note that the subroutine at 1000 moves wasp J1, and the one at 3000 actuates the hunter.

Lines 500 to 570 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 XX and YY. 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).

The test in 1045 is included because the wasp-moving subroutine must be prevented from moving newly-killed wasps.

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 70. 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 4140 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, or for the game to end if the hunter's lives have been used up.

Lines 5000 to 5100 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.

```
  1 REM WASPSHOOTER COPYRIGHT
    (C) ANDREW COLIN 1983
 10 DIMV%(8),U%(8),D%(8,2),W%(20,2)
 20 PRINT" CTRL and 2 SHIFT
    and CLR HOME CRSR
    WASPSHOOTER":PRINT:PRINT
 25 PRINT"COPYRIGHT (C) ANDREW
    COLIN 1983"
 28 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
 95 XX$=" CTRL and 3
    SHIFT and CLR HOME 3 times"
100 SR=1024:PA=54272:FORJ=0TO23:
    POKEPA+J,0:NEXTJ
110 POKE PA+2,100:POKEPA+5,15:
    POKEPA+24,15:POKE PA+8,100:
    POKE PA+12,15:IP=70
115 POKE PA+19,15:POKEPA+6,224:
    POKEPA+4,0:POKEPA+4,65
118 POKE 53281,0:POKE 650,128
120 INPUT"HOW MANY WASPS";N
130 IF N<1 OR N>20 THEN PRINT "1 TO
    20 PLEASE":GOTO 120
150 FOR J = 1 TO 8
160 READ V%(J),U%(J),D%(J,1),D%(J,2)
170 NEXT J
180 DATA -40,93,0,-1,-39,78,1,-1
190 DATA 1,67,1,0,41,77,1,1
200 DATA 40,93,0,1,39,78,-1,1
210 DATA -1,67,-1,0,-41,77,-1,-1
220 BU=INT(7*SQR(N)):SQ=0
230 PRINT" SHIFT and CLR HOME "
240 FOR J=55296 TO 56295 :
    POKEJ,7:NEXT J
250 FORJ=1TON
260 W%(J,1)=INT(40*RND(0))
270 W%(J,2)=INT(12*RND(0))+1
280 POKE SR+40*W%(J,2)+W%(J,1),35
290 NEXTJ
300 NA=N
310 TS=TI
320 A=3:B=18:C=2:GOSUB2000
330 IFNA=0THEN500
335 IFBU=0THEN600
340 PRINT" CTRL and 8 CLR HOME "
```

```
350 PRINT" [CLR HOME] WASPS";NA;"TIME";
    INT((TI-TS)/60);"SHOTS";BU;
    "LIVES";XX$
370 FORJ1=1TONA
380 IFW%(J1,1)>=0THENGOSUB1000
390 GETA$:IFA$="" "THEN410
395 POKE 198,0
400 GOSUB3000
410 NEXTJ1
420 GOSUB2000:GOTO330
500 REM WINS
510 PRINT" [SHIFT] and [CLR HOME] [CRSR ↓]
    [CRSR ↑] WELL DONE !":PRINT
520 PRINT" YOU HAVE KILLED ";N-NA:
    PRINT
530 PRINT"WASPS IN"INT((TI-TS)/60);
    "SECONDS":PRINT
540 PRINT"YOU WERE STUNG":PRINT
550 PRINTSQ;" TIMES"
560 POKEPA+24,0:RESTORE
565 FOR TT=1 TO 5000:NEXT TT
570 PRINT:PRINT" TO HAVE ANOTHER
    GAME HIT ANY KEY"
580 GETA$:IFA$="" " THEN 580
590 GOTO 20
600 REM OUT OF FLY-SPRAY
610 PRINT" [SHIFT] and [CLR HOME] [CRSR ↓]
    [CRSR ↑] SORRY—NO FLY-SPRAY":PRINT
620 PRINT"LEFT!":PRINT
630 GOTO520
1000 REM MOVE J1'TH WASP AT RANDOM
1010 XX=W%(J1,1):YY=W%(J1,2)
1020 XN=XX+INT(3*RND(0))-1:IFXN<0
     OR XN>39 THEN 1020
1030 YN=YY+INT(3*RND(0))-1:IF YN<1
     OR YN>24 THEN GOTO1030
1040 POKESR+40*YY+XX, 32
1045 IF J1>NA THEN 1070
1050 ZZ=SR+40*YN+XN:IFPEEK(ZZ)=32
     THEN POKE ZZ,35
1060 W%(J1,1)=XN:W%(J1,2)=YN
1070 IP=IP+INT(3*RND(0))-1: IF IP<60
     OR IP>80 THEN IP=70
1080 IFXN=A AND YN=BTHENGOSUB
     4000
1090 POKEPA,64*(IP AND 3):POKE
     PA+1,IP/4:RETURN
2000 REM DISPLAY HUNTER
2010 XX=SR+40*B+A:YY=XX+V%(C)
2020 POKEXX,81:POKEYY,U%(C):RETURN
2500 REM ERASE HUNTER
2510 XX=SR+40*B+A:YY=XX+V%(C)
2520 POKEXX,32:POKEYY,32:RETURN
3000 REM MOVE HUNTER OR SHOOT
3010 IFA$<>"M"THEN3040
3020 GOSUB2500:C=C+1:IFC=9THEN
     C=1
3030 GOSUB2000:RETURN
3040 IFA$<>"B"THEN 3070
3050 GOSUB2500:C=C-1:IFC=0THEN
     C=8
3060 GOSUB2000:RETURN
3070 IFA$<>"F"THEN 3150
3080 GOSUB2500:AA=A+D%(C,1):
     BB=B+D%(C,2)
3090 IF AA>2 AND AA<36 AND BB>2
     AND BB<22 THEN A=AA:B=BB
3100 GOSUB2000
3110 FORJJ=1TONA
3120 IFA=W%(JJ,1)ANDB=W%(JJ,2)
     THENGOSUB4000
3130 NEXTJJ
3140 RETURN
3150 IFA$<>"S"THEN RETURN
3160 PP=A+2*D%(C,1):QQ=B+2*D%
     (C,2)
3170 POKEPA+11,129
3180 BU=BU-1
3190 RR=SR+40*QQ+PP
3200 FORKK=1TO5
3210 POKERR,102:FORTT=1TO30:
     NEXTTT
3220 POKERR,32:FORTT=1TO50:
     NEXTTT
3230 NEXTKK
3240 POKEPA+11,0
3250 FORJJ=1TONA
3260 IFPP=W%(JJ,1) ANDQQ=W%(JJ,2)
     THENJ2=JJ:GOSUB5000
3270 NEXTJJ:RETURN
4000 REM HUNTER IS STUNG
4010 PRINT" [CTRL] and [8 (] [CLR HOME]
     S T U N G ! ! ! "
4015 XX$=LEFT$(XX$,LEN(XX$)-1)
4020 GOSUB2500
4030 A=INT(3+16*RND(0)):
     B=INT(3+16*RND(0)):
     C=INT(1+8*RND(0))
4040 POKEPD,15:GOSUB2000:
     SQ=SQ+1
4050 POKEPA+18,17
4060 FORJJ=128 TO 255 STEP 3
4070 POKE PA+15,256-JJ:POKE 53281,
     JJ:POKE 53280,JJ-1
4080 NEXT JJ
4085 IF LEN(XX$)=1 THEN 4100:REM JUMP
     IF LIVES USED UP
4090 POKE PA+18,0:POKE 53281,0:
     RETURN
4100 POKE PA+24,0
4110 PRINT" [SHIFT] and [CLR HOME] [CRSR ↓] [CRSR ↓]
     [CRSR ↓] [CRSR ←] [CRSR →] [CRSR ←] [CRSR →] [CTRL] and
     [" 2] YOU HAVE BEEN STUNG
     THREE TIMES"
4120 PRINT " [CRSR ↓] [CRSR ↓] [CRSR →] [CRSR →] [CRSR →]
     AND YOUR CONSTITUTION CAN
     NO LONGER"
4130 PRINT " [CRSR ↓] [CRSR ↓] [CRSR →] [CRSR →] [CRSR →]
     STAND IT"
4135 PRINT
4140 GOTO520
```

```
5000 REM WASP IS KILLED
5010 PRINT" CTRL and // 2 CLR HOME A
     WASP BITES THE DUST!"
5020 FOR JJ=4 TO 92 STEP 4: POKE
     PA+15,100-JJ
5030 POKE PA+18,17
5040 FORTT=1TO10:NEXTTT
5050 POKE PA+18,0:FORTT=1TO10:
     NEXTTT
5060 NEXTJJ
5070 IFJ2=NATHENNA=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
```


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	

UNIT:24

MORE ABOUT LOGICAL OPERATORS

In this Unit we complete our study of 64 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 7
```

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 64 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 64 (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 64 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 THE 64 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 64.

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 64 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 64, 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 33 129 144
2 50 34 149 5
3 51 35 150 28
4 52 36 151 159
5 53 37 152 156
6 54 38 153 30
7 55 39 154 31
8 56 40 155 158
9 57 41 41 18
0 48 48 48 146
+ 43 219 166 —
– 45 221 220 —
£ 92 169 168 28
CLR HOME 19 147 147 —
INST DEL 20 148 148 —
F1/F2 133 137 137 —
CTRL
Q 81 209 171 17
W 87 215 179 23
E 69 197 177 5
R 82 210 178 18
T 84 212 163 20
Y 89 217 183 25
U 85 213 184 21
I 73 201 162 9
O 79 207 185 15
P 80 208 175 16
@ 64 186 164 —
★ 42 192 223 —
↑ 94 222 222 30
RESTORE
F3/F4 134 138 138 —
RUN STOP 3 131 131 —
SHIFT LOCK
A 65 193 176 1
S 83 211 174 19
D 68 196 172 4
F 70 198 187 6
G 71 199 165 7
H 72 200 180 8
J 74 202 181 10
K 75 203 161 11
L 76 204 182 12
: 58 91 91 27
; 59 93 93 29
= 61 61 61 31
RETURN 13 141 141 —
F5/F6 135 139 139 —
C=
SHIFT
Z 90 218 173 26
X 88 216 189 24
C 67 195 188 3
V 86 214 190 22
B 66 194 191 2
N 78 206 170 14
M 77 205 167 13
, 44 60 60 —
. 46 62 62 —
/ 47 63 63 —
SHIFT
CRSR ↑ ↓ 17 145 145 —
CRSR ← → 29 157 157 —
F7/F8 136 140 140 —
SPACE 32 160 160 —

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.

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:

```
10 GET A$:IF A$ = " " THEN 10
20 PRINT A$; ASC(A$)
30 GOTO 10
```

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 20.

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 90 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:

```
 10 DIM T(26)
 20 GET X$:IF X$=""THEN 20
 30 IF X$="★" THEN 90
 40 PRINT X$;
 50 IF X$ < "A" OR X$ > "Z" THEN 20
 60 P = ASC(X$)-64
 70 T(P) = T(P)+1
 80 GOTO 20
 90 PRINT
100 FOR P = 1 TO 26
110 PRINT T(P);
120 NEXT P
130 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 INST 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 INST DEL and RETURN are detected by their ASCII codes. The INST 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.

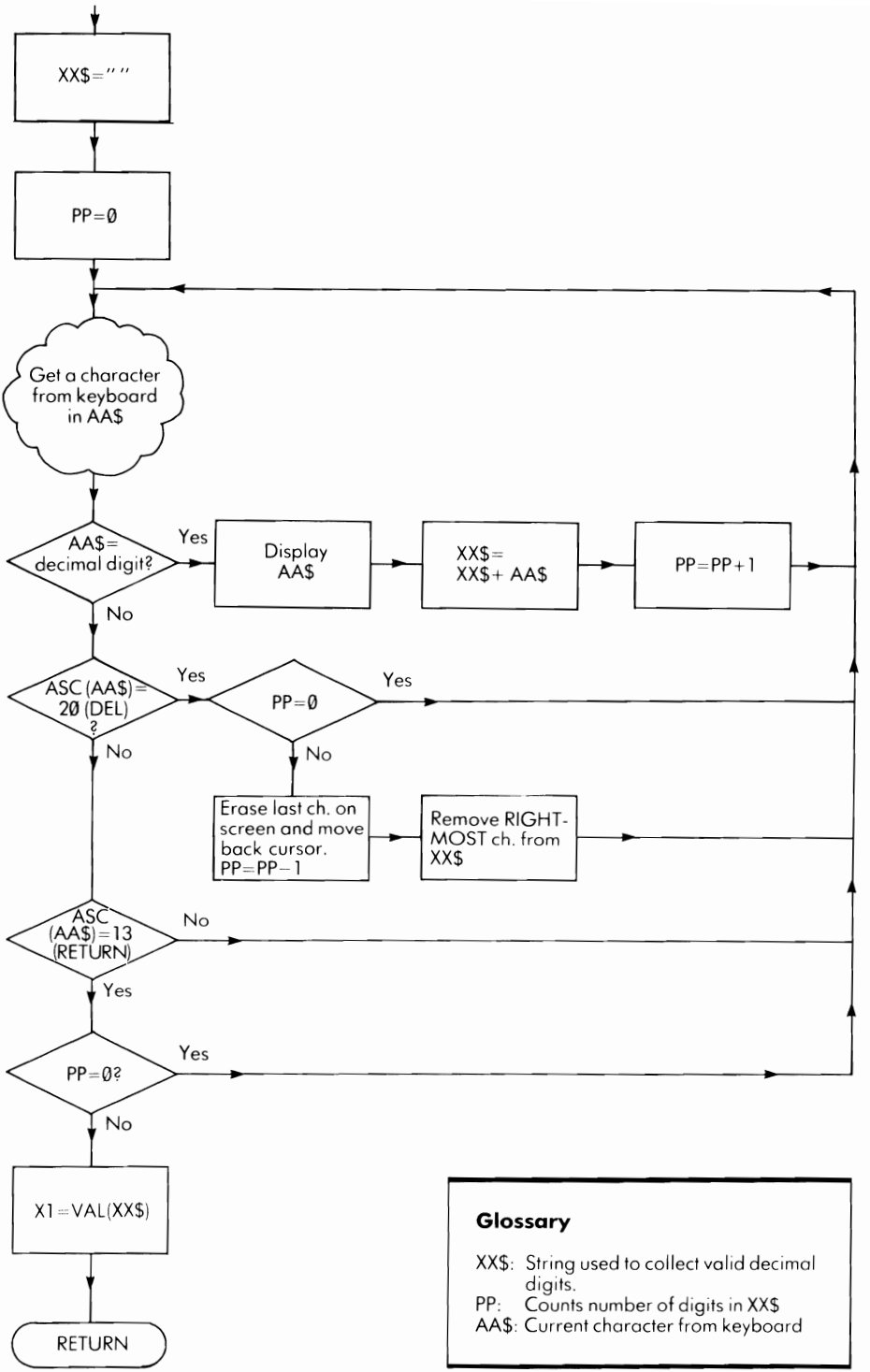
Subroutine Specification

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

```
7000 REM ROBUST NUMBER INPUT
7010 XX$ = "": PP=0
7020 GET AA$: IF AA$ = "" THEN 7020
7030 IF AA$ >= "0" AND AA$ <= "9"
     THEN PRINT AA$;: XX$ = XX$ + AA$:
     PP = PP+1 : GOTO 7020
7040 IF ASC(AA$) <> 20 THEN 7070 : REM
     CHECK FOR DEL (=20)
7050 IF PP = 0 THEN 7020 : REM CAN'T
     DELETE NOTHING
7060 PRINT " SHIFT and CRSR SPACE
     SHIFT and CRSR " ;: PP = PP-1 :
     XX$ = LEFT$(XX$,PP):GOTO 7020
7070 IF ASC(AA$) <> 13 THEN 7020 :REM
     LOOK FOR RETURN
7080 IF PP = 0 THEN 7020 :REM MUST BE
     SOME DIGITS
7090 X1 = VAL(XX$) : 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
000	0-31	——	Control characters
001	32-63	x01	
010	64-95	x00	
011	96-127	——	Not used
100	128-159	——	Control characters
101	160-191	x11	
110	192-223	x10	
111	224-255	——	Not used

In the screen code, x=0 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) +0.5*(AA AND 128):
   IF (AA AND 64)=0 THEN SS=SS+32
```

b) From Screen code to CBM ASCII:

```
AA=(SS AND 31) + 2*(SS AND 64) -
   (SS AND 32)+64
```

Finally, note that CBM ASCII differs in some important details from the standard ASCII as used on many other machines. If you ever come to link up a 64 to another computer, watch this point carefully!

EXPERIMENT 24·1

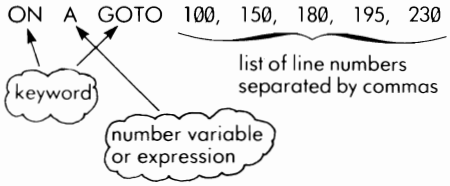
Write a program which allows the user to make simple thick-line drawings. Initially, your program displays a black reversed space in the middle of the screen. This is the beginning of a continuous line which is extended one space upward when the user hits the F1 function key. Similarly, the line is extended 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.



The COMMODORE 64 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 100
IF A = 2 THEN 150
IF A = 3 THEN 180
IF A = 4 THEN 195
IF A = 5 THEN 230
```

If A is less than 1 or greater than 6, the line following the ON will be executed next.

In 64 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:

```
 10 PRINT "DO YOU WANT ADVICE ON"
 20 PRINT "STORING PROGRAMS (1)"
 30 PRINT "USING RND (2)"
 40 PRINT "DRAWING PICTURES (3)"
 50 PRINT "THE ASCII CODE (4)"
 60 PRINT "SOUND PRODUCTION (5)"
 70 INPUT "TYPE 1-5";X
 80 ON X GOSUB 300,400,500,600,700
 90 GOTO 10
...
300 REM GIVES ADVICE ON STORING
    PROGRAMS
...
390 RETURN
400 REM GIVES ADVICE ON USING RND
...
490 RETURN
...
...
700 REM GIVES ADVICE ON SOUND
    PRODUCTION
...
790 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

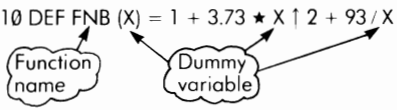
The next feature to be described, the DEF facility, is frankly one of the least useful parts of 64 BASIC. It is suggested that unless you're a good mathematician and specially interested in formulas you skip straight to the next section on storing and retrieving data.

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. For example,

```
FNA or FNX or FNQC or FNG1
```

are all proper formula names.

A formula definition might read:



Once a function has been put into a program it can be used by writing its name with a suitable argument. Thus

```
20 Q = FNB (77)
```

will serve instead of

```
20 Q = 1 + 3.73 ★ 77↑ 2 + 93/77
```

or `30 PRINT FNB(S)`

can be used for

```
30 PRINT 1 + 3.73 ★ S ↑ 2 + 93/S
```

or again,

```
40 ZZ = FNB(P-Q)
```

is now a valid way of writing

```
40 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 64 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 − 100

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 0.1. 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

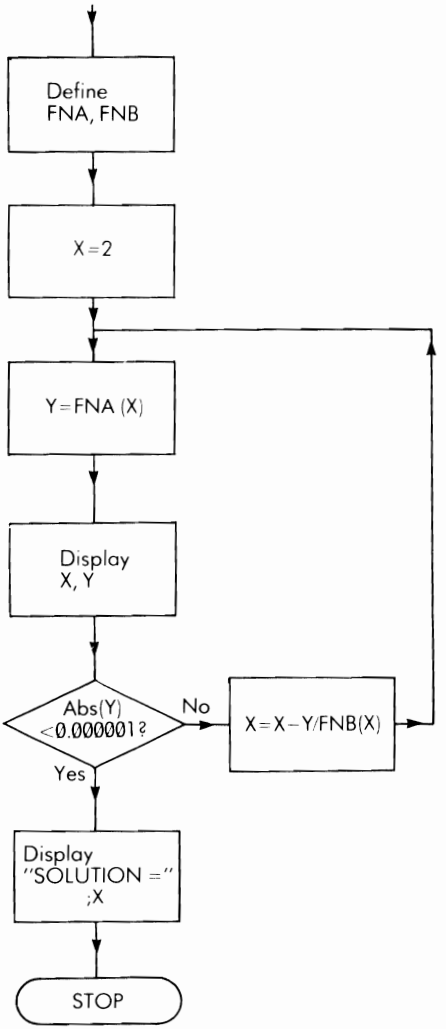
3 ★ X ↑ 2 + 2 ★ (X + 7)

(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

Non-mathematical readers rejoin here! Next, we look at commands for storing and retrieving data (as opposed to programs) on cassette tape or diskette. For example, you may have a large collection of scientific observations or replies to a questionnaire which must be analysed with several different programs: one to plot graphs, one to work out statistics, and so on. Clearly it is worth keeping this data in "machine-readable" form on the backing store so that you don't have to type it in over and over again.

The basic principles of using cassettes and diskettes are quite similar but the practical details are so different that it would be confusing to consider both at the same time. Instead, there is one section about cassettes and another separate one on diskettes. You should study the section which applies to your computer and miss the other one out for the present.

DATA ON CASSETTE

The basic unit of storage on a cassette tape is the *file*. It is arranged in three sections:

HEADER	FILE BODY	TRAILER
file name	characters	end of file marker

← 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 cassette side (45 minutes). Each minute of playing time holds about 1200 characters.

The file body is divided into equally sized 'blocks' each holding about 192 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 recording system works automatically.

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.

PRINT# TO WRITE DATA

To write a file on to a cassette we use three new commands:

```
OPEN 1,1,1,"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 1 must be precisely as they are shown.

When the OPEN is executed, the 64 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 64 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!

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 80 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 80.

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 64 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 64 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 1,1,0,"file name"
INPUT # 1,
CLOSE1
```

The OPEN command with the zero in front of the file name (instead of a 1) makes the 64 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 64 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 1,1,0,"THURSDAY DATA"
or  OPEN 1,1,0,"THURSDAY"
or  OPEN 1,1,0,"T"
or  OPEN 1,1,0,"" : REM NULL STRING WILL
          OPEN ANY FILE
or  OPEN 1,1,0 : REM TITLE CAN BE OMITTED
or  X$ = "THURS" : OPEN 1,1,0,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 64 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 0 means that all is well. The value 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 5000 : NEXT S
90 INPUT" SHIFT and CLR HOME ";X$ :
   REM USER DRAWS PICTURE

100 FOR J = 0 TO 999 : REM SCAN SCREEN
    AND COLOUR RAMS
110 PRINT # 1, PEEK(1024 + J);",";PEEK
    (55296+J)
120 NEXT J
130 CLOSE1
140 STOP
```

now rewind the tape and type in the following program:

```
10 OPEN 1,1,0,"SCREEN PICTURE"
20 J=0
30 INPUT # 1,X,Y : REM GET SCREEN AND
   COLOUR CODES
40 POKE 1024+J,X : POKE 55296+J,Y
50 IF ST <> 0 THEN 70
60 J = J+1 : GOTO 30
70 IF ST = 64 THEN 100 : REM CHECK FOR
   END OF FILE
80 PRINT "TAPE FAULT"
90 STOP
100 CLOSE 1
110 GOTO 110 : REM LOOP STOP IF ALL
    WELL
```


GET

Another command which is sometimes used with cassette tapes is

```
GET # 1,
```

This command is rather like INPUT # 1, except that it transfers single characters. It would appear in sequences like

```
...
100 GET A$ : IF A$ = "" THEN 100 : REM
    GET A CH. FROM KEYBOARD
110 PRINT A$:REM DISPLAY IT
120 PRINT # 1, A$:REM SEND IT TO
    CASSETTE TAPE
...
```

and

```
...
200 GET # 1,X$: REM GET A CHARACTER
    FROM TAPE
210 IF ST <> 0 THEN 300 :REM JUMP IF END
    OF FILE OR ERROR
220 PRINT X$;
...
```

PROBLEMS

As you have seen, the 64 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 may be higher than you get with programs, because

a) Data files are generally longer

b) There is no way of 'verifying' data files like programs

3) The 64 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 64's memory.

If you are seriously interested in storing large amounts of data, you should invest in a disk drive. **Commodore has a very good unit specifically** designed for the 64. 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.

DATA ON DISKETTES

If you have a disk drive for the COMMODORE 64, you will be familiar with the most important aspects of using diskettes. You'll know how to take good care of the disks themselves, how to format and initialise them, and how to load and store programs.

In this section we shall discuss ways of using diskettes to store data rather than programs. The disk drive is by its nature a complex device, and we must begin by filling in some important background information.

As you already know, every disk command includes some mysterious numbers. For example, you always type

```
LOAD "MYPOGRAM",8 or
OPEN 1,8,15,"I"
```

The time has come to explain what these numbers mean.

Whenever any peripheral device is connected to the computer, there are three basic ideas which have to be understood.

First, we have the idea of a channel number (sometimes rather misleadingly called a "logical file number", since it doesn't always refer to a file, and isn't more logical than any other kind of number). A computer could be likened to a travel agent sitting behind a desk. He can receive information from all sorts of different sources: the customer across the table, two or three telephones, and a terminal connected to some distant computer. He can also send messages in different ways: for example, by speaking them to his customer or typing them into the computer console.

In the 64 analogy, each one of these sources or destinations of information would be called a channel. For example, the customer might be channel number 1, the computer terminal channel number 2, and the telephones channels 3 to 5.

When a program needs to communicate with a peripheral device, it must first set up an appropriate channel. A BASIC program can use a large number of channels to various devices (up to 10). They can be numbered in any way you like, provided that the channel numbers are all different and lie in the range 1 to 127.

Second, we have the idea of a device number. Every unit connected to the 64 has its own fixed device number which is built in at the factory and can't easily be changed. For example, the device numbers of the tape cassette, the printer and the disk drive are always 1, 4 and 8 respectively.

At this point you may ask, why not simply make the channel number the same as the device number and have done with it? The reason is that such an arrangement doesn't give you enough flexibility. As we shall see, there are quite often more channels than devices!

The third important concept is the 'secondary address'. In some types of device, an information channel can be used in different ways; for example, a tape cassette recorder can either read or write, but not both at the same time. The 'secondary address' for a channel indicates exactly how the channel is to be used. Note that the meaning of secondary addresses such as 0 or 1 is not fixed but depends on the actual device it refers to.

When a BASIC program is running, we can imagine each channel to be a telephone wire running to or from the main computer. Figure 24.2 shows the general pattern.

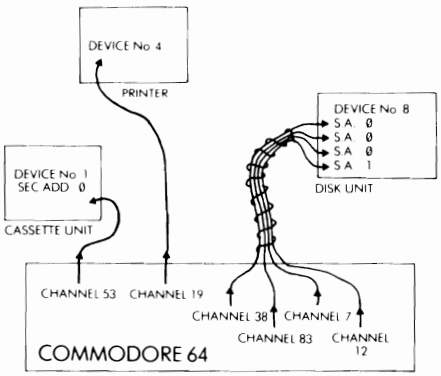


Figure 24.2

You will see that each wire really has two or even three different labels. In the computer the wire is labelled with its channel number. As it comes out it is coded by the device number it goes to. When it arrives at the device it may be labelled yet a third time with a 'secondary address'.
Of course you must not mistake the diagram for an actual picture of the way the peripherals are physically connected to the 64. In practice, all the channels (except the one which connects the cassette recorder) share the same pair of wires, which link all the peripherals in a daisychain arrangement. These wires are called a "serial bus".

THE "OPEN" COMMAND

We can now explain the structure and meaning of such commands as OPEN. The OPEN command sets up a new channel to an external device. The keyword OPEN is always followed by two or three numbers and sometimes a string. The first number gives the channel number, the second is the physical device, and the third is the secondary address. The string, if there is one, is a message which the computer sends to the channel when it is opened.

In some cases, when a channel is to be opened on a device that doesn't use secondary addresses, the OPEN command only needs two numbers. For instance to use the printer (device number 4) as channel number 25, we'd put

```
OPEN 25,4
```

To select a particular secondary address within a device, we need all three numbers after the keyword. For example,

```
OPEN 33,8,5
```

would open channel 33 as secondary address 5 in the disk controller.

Let's now turn our attention to the details of the disk controller.

The disk drive includes 7 secondary addresses, numbered 0 to 5 and 15. Three of them are used for special purposes:

Secondary address 0 is used for LOADING programs.

Secondary address 1 is used for SAVEing programs.

Secondary address 15 is used as the control channel, which handles all the special functions such as loading the directory, formatting and initialising disks, and reporting errors which occur on the other channels. Whenever the diskette is used for any purpose (except simply loading or saving a program) a channel connected to secondary address 15 must be set up. This can be done by a command such as

```
OPEN 1,8,15
```

or

```
OPEN 1,8,15, "I0"
```

Both of these commands specify channel number 1. The second one initialises the diskette as well as opening the channel.

The other 4 secondary addresses (2 to 5) can be used to send data to and from the diskette. To do this, your program must have at least two channels open to the disk drive at the same time — the control channel and one data channel. This is quite normal; in fact, you can open a separate channel for each secondary address. Up to 3 data channels plus the command channel can be open at one time.

The basic unit of storage on the disk is the file. There are two main kinds of file: program files (which can be SAVEd and LOADed), and sequential data files which store items such as numbers and strings. Every file has a name which can be up to 16 characters long and is kept in a directory on a special part of the diskette. A file can be as long as you like provided that all the files on the diskette, when taken together, don't come to more than about 160000 characters.

PRINT# TO WRITE DATA

When your program comes to use the diskette for storing data, you will start by creating a new file. This is how you do it:

First, open a channel to secondary address 15, which handles all the organisational problems. You may as well initialise the diskette at the same time.

Next, open another channel to one of the ordinary secondary addresses (say number 2). The string which follows the secondary address must give the details of the file you want to write. You must quote a file name, a type (which is always SEQ) and a mode which is always WRITE if you are creating a new file. These three items are separated by commas and placed inside string quotes.

To give an example, suppose you wish to set up a data file called DINOSAUR. The program to do the job would start with

```
10 OPEN 1,8,15,"I0" : REM OPEN
   CONTROL CHANNEL AND INITIALISE
20 OPEN 2,8,3,"DINOSAUR,SEQ,WRITE" :
   REM CREATE NEW FILE
```

Note that the control channel number (1) could have been anything between 1 and 127; the data channel number (2) could have had any value in the same range except the one allocated to the control channel, and the secondary address (3) could have been any number between 2 and 5. On the other hand, you have no choice at all about the device number (always 8 for the disk unit) and the control channel (always 15).

Once you've opened a data file, you can use the PRINT# command to send information to be recorded. The PRINT# must be followed by the channel number of the data file, and then by the variables or expressions you want to record. The channel number must be followed by a comma, and the variable names (if there are more than one) by the sequence ;",","; . Some examples are

```
PRINT#2,X
PRINT#2,P$;",";X+5
PRINT#2,Q$;",";S$",";J;",";K
```

Note that PRINT# is not the same keyword as PRINT. In particular you must not write anything like '?#' for PRINT#; it simply won't work. Also there must be no spaces between the PRINT and the # or between the # and the channel number.

You can have as many variables as you like in the PRINT # command, provided that the total

length of the command doesn't exceed 80 characters (the normal limit for commands).

When your program obeys a PRINT# command, it sends the values of the variables mentioned to the file specified by the channel number. In most programs the PRINT# command will be placed inside a loop, so that it is called repeatedly. This way, a file of any length can be built up.

When a program has finished making a new file, it must give a CLOSE command. This takes the form

```
CLOSE n
```


where n is the channel number of the new file. Examples are

```
CLOSE 5
```

or

```
CLOSE 6
```

When your program has finished all communications with the disk controller, it must close the channel to secondary address 15. This is also done with a CLOSE command.

INPUT# TO READ DATA

To get your information back from a diskette, your program must begin by opening a control channel to the disk controller, just as described above. Then it has to open a data channel to the file you want to read. The name of the file must be the same as one you've created previously, and the mode should be READ. The appropriate instructions might be

```
OPEN 2,8,15,"I0" : REM OPEN CONTROL
     CHANNEL
OPEN 5,8,11,"DINOSAUR,SEQ,READ" :
     REM OPEN DINOSAUR FOR
     READING
```

Notice that files do not remember which channel or secondary address they were written on.

The INPUT# command is like PRINT# in reverse. The keyword is followed by the names of the variables to be read from the diskette, separated by commas if there are more than one. Some examples are

```
INPUT#1,Z
INPUT#1,P$
INPUT#1,R$,Q,T$
```

Note that the number and type of variables which follow the keyword must be the same as those used to put values on the diskette in the first place. For instance the command

```
INPUT#4,A$,B$,C :REM FETCH TWO
STRINGS AND A NUMBER
```

could be used to retrieve data originally written by

```
PRINT#2,A$;",";B$;",";C :REM RECORD
TWO STRINGS AND A NUMBER
```

or

```
PRINT#99,Z$(Q);",";"LEADS TO";",";55 :
REM TWO STRINGS AND A NUMBER
```

but not

```
PRINT#22, A;",";B$ : REM A NUMBER AND
A STRING
```

At this point it is worth giving a simple example. Key in the following program, which allows you to preserve a complete screen picture on a file:

```
10 OPEN 1,8,15,"10":REM EYE ZERO
20 OPEN 2,8,3,"SCREEN,SEQ,WRITE"
30 PRINT " SHIFT and CLR HOME , CRSR ↓ "
40 PRINT "DRAW ANY PICTURE YOU LIKE"
50 PRINT "USING THE CURSOR"
60 PRINT "AND COLOUR CONTROLS"
70 PRINT "LEAVE THE CURSOR ON THE"
80 PRINT "TOP LINE, WHICH MUST BE"
90 PRINT "EMPTY. THEN PRESS RETURN"
100 FOR S = 1 TO 5000 : NEXT S
110 INPUT " SHIFT and CLR HOME ";X$
120 FOR J= 0 TO 999 : REM SCAN SCREEN
130 PRINT# 2, PEEK (J+1024);",";PEEK
    (J+55296) : REM WRITE DETAILS OF A
    CHARACTER
140 NEXT J
150 CLOSE 2 :REM CLOSE DATA CHANNEL
160 CLOSE 1 :REM CLOSE CONTROL
    CHANNEL
170 STOP
```

Now load a diskette which has been formatted, and run the program. When it invites you to draw a picture, do so with the cursor control, the colour keys and any graphics symbols you care to choose. Keep the top line of the screen clear.

When you are ready, move the cursor to the top line and press RETURN. The program will then work its way through the screen, finding out the character code and the colour of each character and writing them to a file called SCREEN. This will take a minute or two, during which you will see the disk light glowing.

When the program ends, load the directory from the diskette and list it. Amongst the various programs you have stored you will see a new file called SCREEN. It will be about 40 blocks long (depending on the details of your picture) and will be of type SEQ (not PRG like the others).

To get your picture back, type the following program:

```
10 OPEN 1,8,15,"I0"
20 OPEN 2,8,2,"SCREEN,SEQ,READ"
30 J=0
40 INPUT#2, X,Y
50 POKE 1024+J, X
60 POKE 55296+J,Y : REM RESTORE
   CHARACTER AND COLOUR
70 IF ST <>0 THEN 100
80 J=J+1
90 GOTO 40
100 IF ST = 64 THEN 120
110 PRINT "DISK FAULT"
120 CLOSE 2
130 CLOSE 1
140 GOTO 140
```

When you run this program, it will read the description of your picture back from the file and restore it to the screen. Notice that the file is now opened in a different mode.

When the computer is reading a file, various unexpected things can happen. For example, the diskette might have been damaged so that the information is unreadable, or you might simply have come to the end of the file.

To help with these problems, Commodore have provided the special variable ST (which stands for "STatus"). Whenever the computer does something with a file (such as reading or writing some values) ST is automatically set up to show what happened. The value 0 means that everything went as expected. 64 comes up when you read the last bit or item of the file and 'real' faults are indicated by various other codes. The program you have just tried uses ST to stop reading as soon as the screen is full.

This section has given you only the briefest introduction to the use of files on the diskette. In practice it is the presence of the disk controller which turns the 64 into a serious computer capable of word processing and numerous scientific and business applications. To learn more about the disk drive system, read the VIC 1541 reference manual.

EXPERIMENT 24·3

In this experiment we shall design and build a mechanical Morse Code teacher.

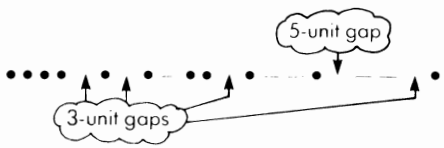
Morse is used to send information by radio. Each letter of the alphabet has a code which consists of dots and dashes. The full code is

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 64 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 0 0 0   (A)
3 1 1 1 0   (B)
3 1 3 1 0   (C)
-----------
3 3 1 1 0   (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 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 type the sentence 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.

FOR CASSETTE OWNERS

Load the program MAKENAMES (T) from the cassette tape. Then put a fresh cassette into the recorder and run the program. It will produce a list of 100 people and write their personal details on to the cassette in a file called "COMPUTER DATES". The program takes about 5 minutes to run, and the screen will be blank for most of the time.

FOR DISK DRIVE OWNERS

Load the program MAKENAMES (D) from the diskette. Then put a formatted diskette (one which is safe to write on) into the disk unit and run the program. It will produce a list of 100 people and write their personal details on to the diskette 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

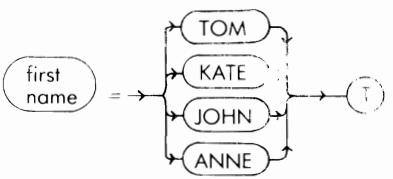
PROGRAM DESIGN — CASE STUDIES

The topic of program design has been a recurring theme throughout this book. This 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.

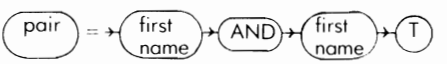
The notation often used for these rules is called a "tramline" grammar. Suppose that at a certain point in a sentence being constructed we need a person's name to be selected from the list TOM, KATE, JOHN, ANNE. We can write down this part of the grammar with the aid of the diagram



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:

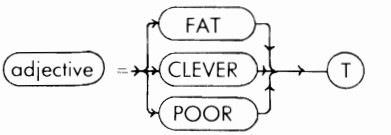


(where "first name" is the diagram with TOM, KATE, JOHN and ANNE).

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



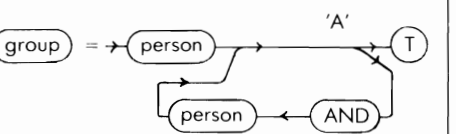
then the grammar

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

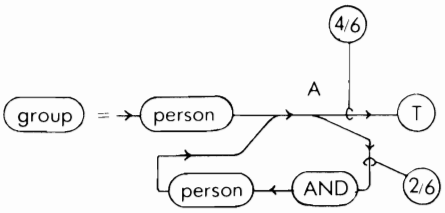
TOM
CLEVER TOM AND KATE
FAT TOM AND POOR ANNE AND CLEVER JOHN
TOM AND JOHN AND POOR KATE
or possibly FAT TOM AND FAT TOM AND FAT TOM

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 64 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(0)))
```

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)

1320 makes a random decision whether to end the phrase

1330 puts in the word (AND)

1340 selects another (person)

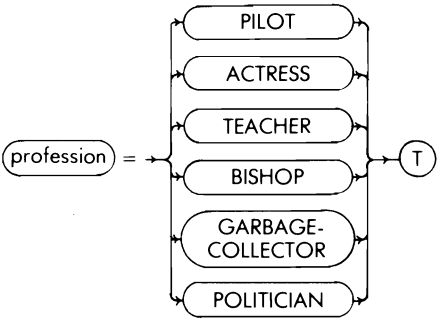
1350 returns the subroutine to the point where it decides whether to stop or go round again.

Finally, we supply some "driver" commands in lines 40, 50 and 60.

```
10 DIM N$(7)
20 N$(1) = "TOM ": N$(2)="KATE ":
   N$(3)="JOHN ":N$(4)="ANNE "
30 N$(5) = "FAT ": N$(6)="CLEVER ":
   N$(7)="POOR "
40 X1$=" "
50 GOSUB 1300
60 PRINT X1$
70 GOTO 40

1000 REM FIRST NAME
1010 X1$=X1$+" "+N$(1+INT(4*RND(0)))
1020 RETURN

1100 REM ADJECTIVE
1110 X1$=X1$+" "+N$(5+INT(3*RND(0)))
1120 RETURN

1200 REM PERSON
1210 IF RND(0) < 3/6 THEN GOSUB 1100:
     REM CALL ADJECTIVE
1220 GOSUB 1000 : REM CALL PERSON
1230 RETURN

1300 REM GROUP
1310 GOSUB 1200 : REM CALL PERSON
1320 IF RND(0) < 4/6 THEN RETURN : REM
     POINT "A" IN DIAGRAM
1330 X1$=X1$+" AND "
1340 GOSUB 1200 : REM CALL ANOTHER
     PERSON
1350 GOTO 1320
```

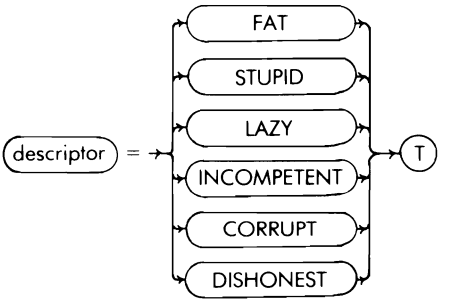
Key in this program and run it. Then try the effect of varying the probability markers in lines 1210 and 1320.

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:



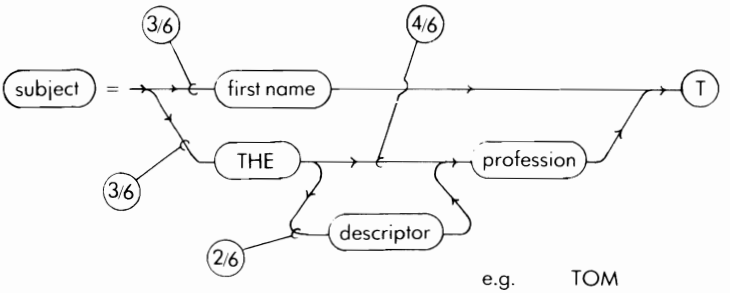
e.g. BISHOP

or .....



e.g. LAZY

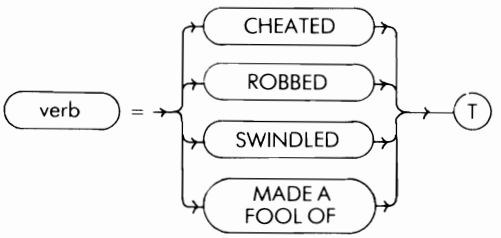
or .....



e.g. TOM

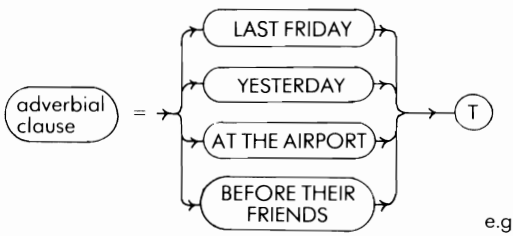
or THE STUPID PILOT

or .....



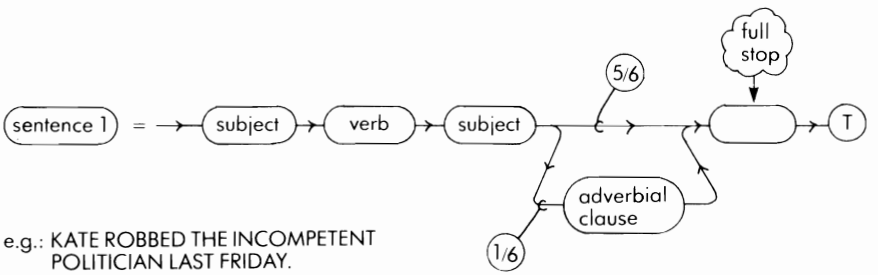
e.g. ROBBED

or .....



e.g. YESTERDAY

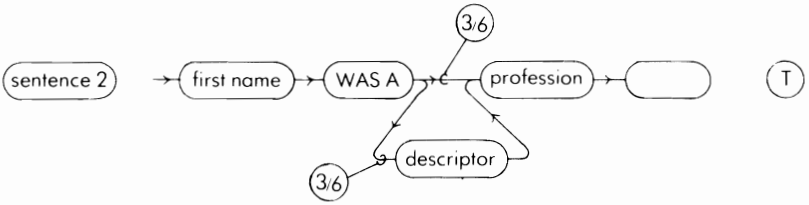
or .....



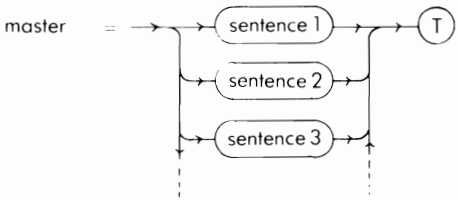
e.g.: KATE ROBBED THE INCOMPETENT POLITICIAN LAST FRIDAY.

or .....

You can have any number of different sentence definitions, such as:



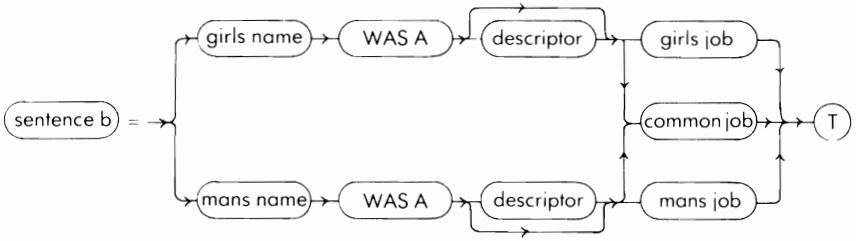
and you can combine them into a 'master' tramline diagram which includes all the sentence forms you want to generate. It would begin:



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 or diskette. 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:

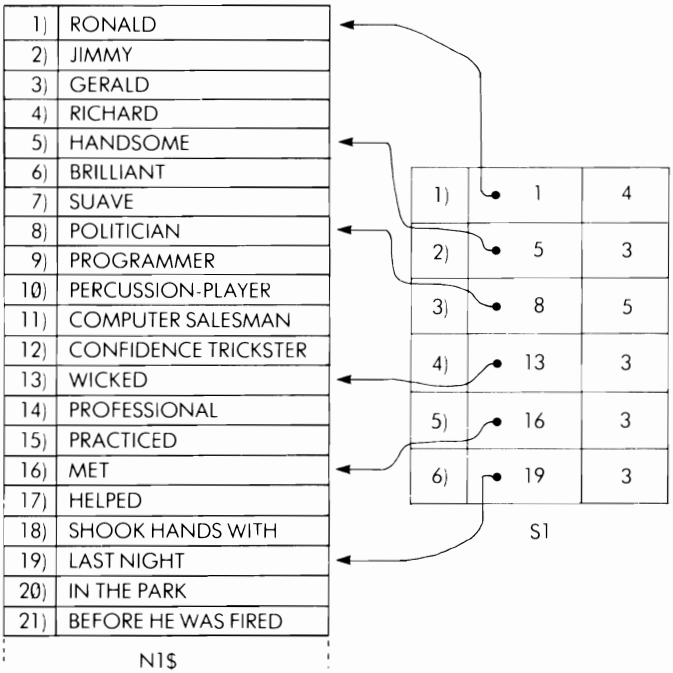
```
10 DATA RONALD,JIMMY,GERALD,
        RICHARD,Z
20 DATA HANDSOME,BRILLIANT,SUAVE,Z
30 DATA POLITICIAN,PROGRAMMER,
        PERCUSSION-PLAYER,
        COMPUTER SALESMAN,
        CONFIDENCE TRICKSTER,Z
40 DATA WICKED,PROFESSIONAL,
        PRACTICED,Z
50 DATA MET,HELPED,SHOOK HANDS
        WITH,Z
60 DATA 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:

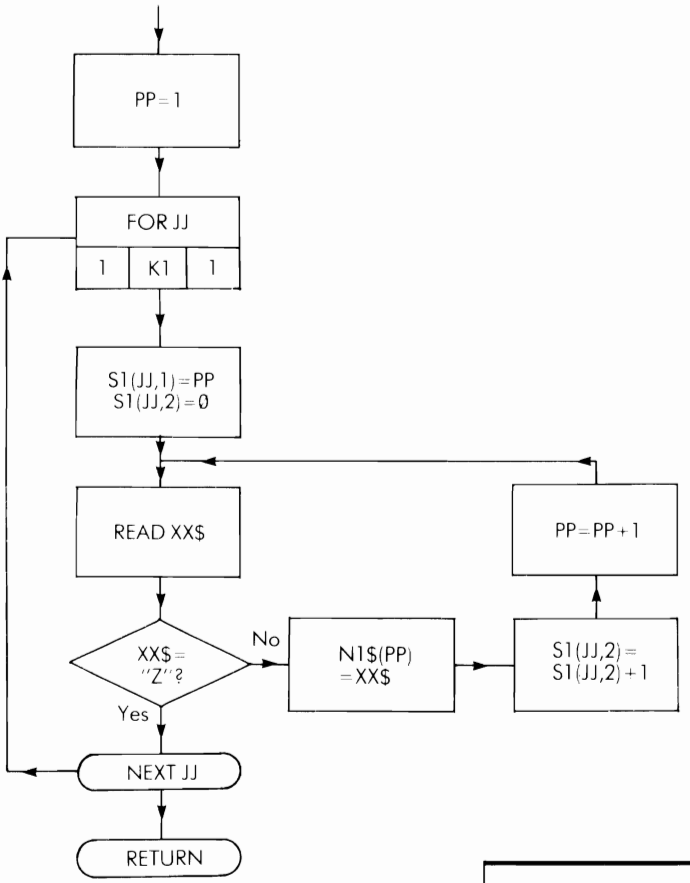


In this data structure, row 3) of array S1 tells the program that group 3 contains 5 words starting at N1$(8). An instruction to add a group 3 word to X1$ would read:

```
X1$=X1$ + " " + N1$(S1(3,1) + INT(S1(3,2) * RND(0)))
                  =8          =5
```

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(sign-posts)

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.

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.

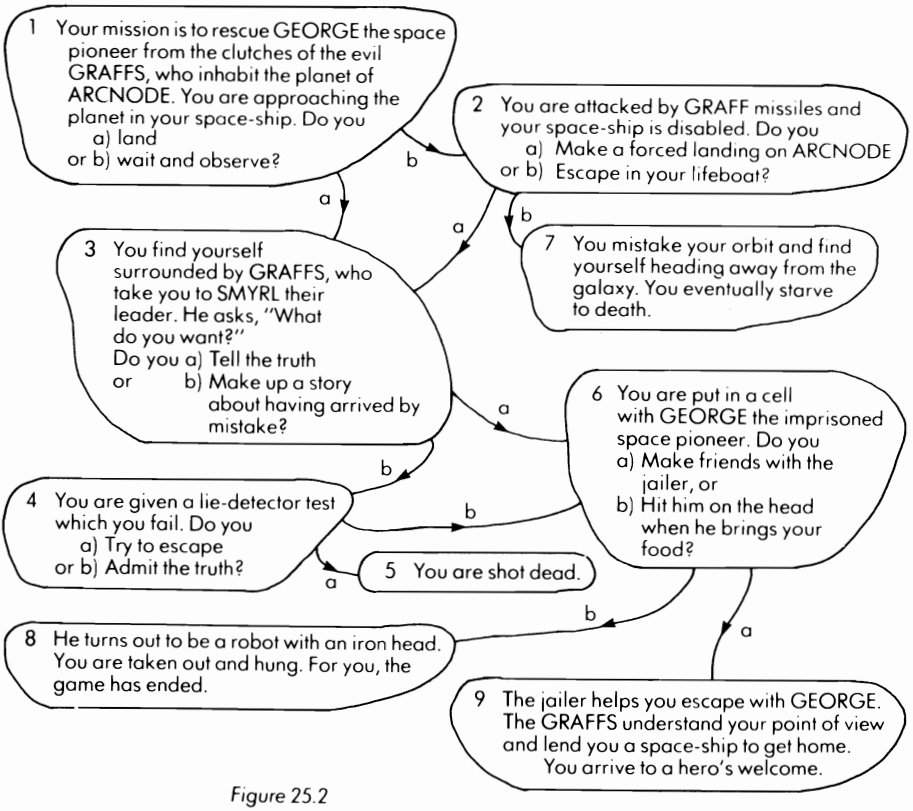


Figure 25.2

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.

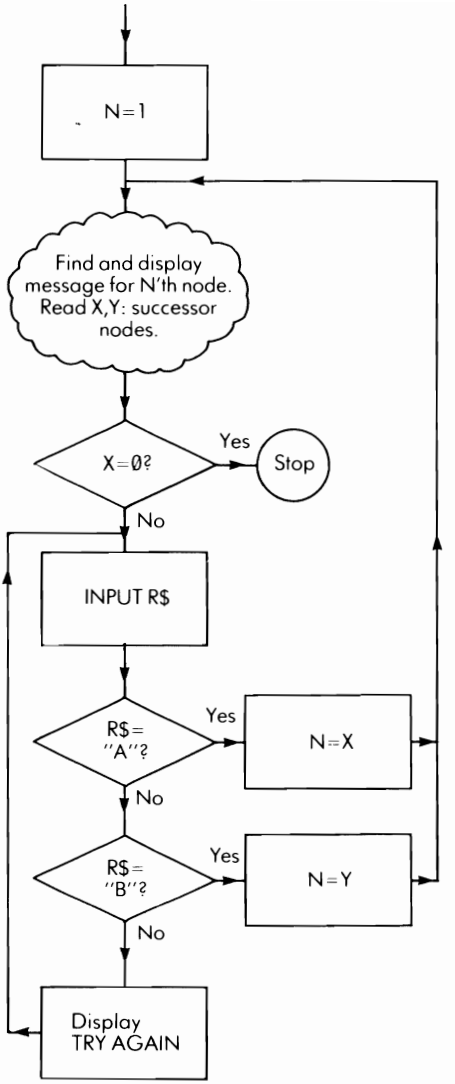
b) It then invites the user to type A or B, and rejects all other inputs.

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 240 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 5500 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.

```
1000 REM PROGRAM BEGINS HERE
1010 N=1 : REM START AT NODE 1
1020 RESTORE : REM FIND N'TH NODE
1030 FOR J = 1 TO N
1040 READ J$,K$,L$,M$,X,Y : REM AND
     READ ITS CONTENTS
1050 NEXT J
1060 X1$=J$ + K$ : GOSUB 5500: REM
     DISPLAY MESSAGE
1070 X1$ = L$ : GOSUB 5500 :REM FIRST
     ALTERNATIVE
1080 X1$ = M$ : GOSUB 5500 :REM
     SECOND ALTERNATIVE
1090 IF X = 0 THEN STOP : REM END OF
     GAME
1100 INPUT R$ : REM GET REPLY
1110 IF R$ = "A" THEN N=X : GOTO 1020
1120 IF R$ = "B" THEN N=Y : GOTO 1020
1130 PRINT "TRY AGAIN": GOTO 1100
```

followed by the commands of the subroutine at 5500.

EXPERIMENT 25·2

Load the full program of GRAFFS and try it out for yourself. Then modify it:

a) To produce interesting sound effects

b) To tell a different story-line.

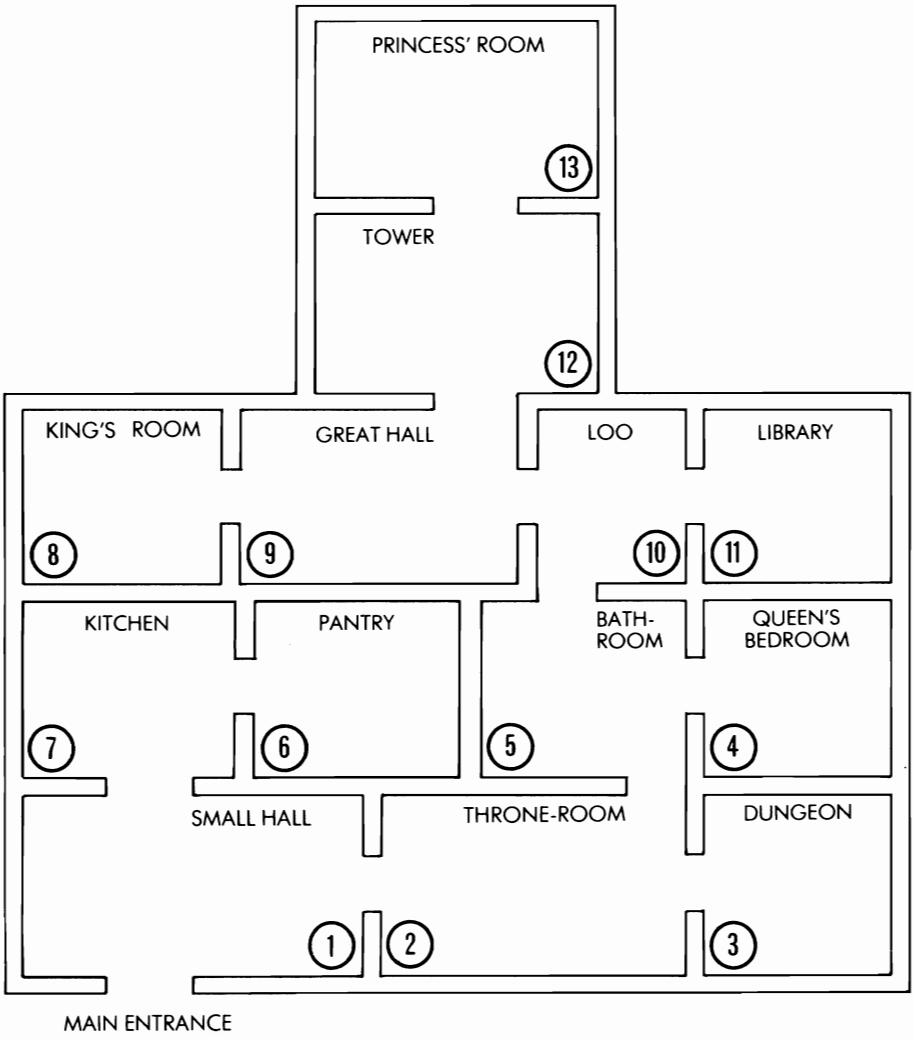
Experiment 25.2 Completed

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 must be short enough for you to study in detail.

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	0
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	10
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:
"a Dragon"
"You fight and"
"kill it with your sword"
"it kills you"
"you both run away!"
The battle subroutine (lines 3000 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 5,3,0,1. This means that the Throne-room (2) is connected to the bathroom (5) going North, the Dungeon (3) going East, and the Small Hall (1) going West. 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. 0 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	

UNIT: 26

SPRITES

This unit is about the design and use of Sprites. A Sprite is a small picture which moves about the screen of your machine smoothly and independently of anything else which may be displayed at the same time. As you may remember, the demonstration program which went with Part 1 of this Series showed you pictures of fireworks, balloons and parachutes. They were all examples of Sprites.

The Sprite facility lets you present vivid animated pictures to illustrate your program. Sprites are invaluable in many areas: games, educational programs and industrial monitoring and control.

Let's begin with a word of caution. Sprites are not easy to manage. They are not especially difficult either, but you will have to exercise your full artistic abilities as well as care and patience in programming if you want a really impressive display.

EXPERIMENT 26·1

To start, load and run the Sprite Display Program, SDP. Try out some of the things it can do:

a) When you press the cursor keys (and SHIFT) in the normal manner, the fish moves up and down, left and right. It can even move right off the screen. You can select the speed of movement by typing F ("Fast") or S ("Slow"). Notice that when the fish is moving slowly, it also moves smoothly — not in jerks which correspond to characters or lines.

b) The fish will change size and shape when you type ↑ or ←. In this program these keys have been arranged to be double-acting: that is, the second hit will reverse the effect of the first one.

c) The eight colour keys, when used together with the CTRL or C= key, will turn the fish into any one of sixteen colours. One of them, light grey, is invisible against the background colour, so a warning message appears because the fish is perfectly camouflaged.

d) If you move the fish to part of the screen where the instructions are displayed, you'll see it apparently go behind the letters, so they remain visible. Now hit the £ key. This brings the fish forward, so that it is in front of the characters and hides them. Hit £ key again and the fish moves back.

This is a demonstration of "Data-Sprite Priority". The Sprites and the characters displayed are normally independent, but if both objects are to be shown in the same place the computer has to decide which to put in front. Fortunately this decision is under the control of the program. Think for a moment, and predict what will happen if you repeat this experiment with the fish in its 'invisible' colour. Then try it and see if you were right.

Experiment 26.1 Completed	

The 'fish' program introduced you to some of the simpler properties of Sprites. A full listing of the program is given below, and you should refer to it while studying the Unit.

```
10 REM COPYRIGHT (C) ANDREW COLIN
   1983
15 REM
20 REM DEFINITION OF FISH SPRITE
30 DATA0,8,0,0,48,0,0,192,0
40 DATA3,128,3,31,192,28,115,248,112
50 DATA243,255,224,255,255,192,127,
   255,224
60 DATA31,248,112,7,224,28,1,192,3
70 DATA0,224,0,0,24,0,0,4,0
80 DATA0,0,0,0,0,0,0,0,0
90 DATA0,0,0,0,0,0,0,0,0
100 REM
110 REM SET UP SPRITE DATA
120 FORJ=832 TO 894
130 READ A: POKE J,A
140 NEXT J
150 REM
160 REM SET V = ADDRESS OF SPRITE
    CONTROLLER
170 V=208*256
180 REM DISPLAY INSTRUCTIONS
185 POKE 53281,15
190 PRINT " [SHIFT] and [CLR HOME] [CTRL]
    and [!1] [CRSR] "
200 PRINT"    SPRITE DEMONSTRATION
    PROGRAM"
205 PRINT"    ====== ========
    ===== ======="
210 PRINT:PRINT
220 PRINT" USE CURSOR CONTROLS TO
    MOVE"
225 PRINT" THE FISH —"
230 PRINT
240 PRINT" COLOUR KEYS"
245 PRINT" (AND CRTL OR COMMODORE
    KEY)"
250 PRINT" TO CHANGE ITS COLOUR "
260 PRINT
270 PRINT" ↑ AND ← TO CHANGE ITS
    SHAPE"
271 PRINT:PRINT" F OR S TO MAKE IT
    MOVE "
272 PRINT" FAST OR SLOW"
275 PRINT:PRINT" AND £ TO PUT IT IN
    FRONT OF"
278 PRINT" OR BEHIND THE TEXT ON THE
    SCREEN"
280 REM
290 REM SET UP THE SPRITE FOR DISPLAY
300 POKE 2040,13 : REM SET POINTER
    TO SPRITE DEFINITION (13*64=832)
310 X=200:Y=230 :R=1:REM SET SPRITE
    POSITION AND RATE
315 POKE V+39,1:POKE V+27,1:REM
    SELECT COLOUR WHITE; SPRITE IN
    FRONT OF TEXT
320 POKE V+23,0:POKE V+29,0:REM
    SELECT SMALL DIMENSIONS
330 POKE V+21,1:REM ENABLE SPRITE
340 REM UPDATE SPRITE POSITION AND
    READ KEYBOARD
350 POKE V,X AND 255:REM LOAD
    HORIZONTAL POSITION
360 POKE V+16,INT(X/256)
370 POKE V+1,Y: REM SET VERTICAL
    POSITION
380 GET A$:IF A$=" "THEN 380
385 PRINT" [CLR HOME] [SPACE]    30 times"
390 REM ANALYSE KEY PRESSED
400 IF A$ = " [CRSR] " AND X<343 THEN
    X=X+R:GOTO340:REM MOVE RIGHT
410 IF A$ = " [SHIFT] and [CRSR] " AND
    X>0 THEN X=X-R:GOTO 340:REM
    MOVE LEFT
420 IF A$ = " [CRSR] " AND Y<249 THEN
    Y=Y+R:GOTO340:REM MOVE
    DOWN
440 IF A$ = " [SHIFT] and [CRSR] " AND
    Y>8 THEN Y=Y-R:GOTO340:REM
    MOVE UP
450 IF A$ = "↑" THEN POKE V+23,1-PEEK
    (V+23):GOTO340 : REM CHANGE
    VERTICAL SIZE
460 IF A$ = "←" THEN POKE
V+29,1-PEEK
    (V+29):GOTO340 : REM CHANGE
    HORIZONTAL SIZE
470 REM NOW TEST COLOUR KEYS
480 IF A$ = " [CTRL] and [!1] "THEN
    POKE V+39,0:GOTO340:REM BLACK
490 IF A$ = " [CTRL] and ["2] "THEN
    POKE V+39,1:GOTO340:REM WHITE
500 IF A$ = " [CTRL] and [#3] "THEN
    POKE V+39,2:GOTO340:REM RED
510 IF A$ = " [CTRL] and [$4] "THEN
    POKE V+39,3:GOTO340:REM CYAN
520 IF A$ = " [CTRL] and [%5] "THEN
    POKE V+39,4:GOTO340:REM PURPLE
530 IF A$ = " [CTRL] and [&6] "THEN
    POKE V+39,5:GOTO340:REM GREEN
540 IF A$ = " [CTRL] and ['7] "THEN
    POKE V+39,6:GOTO340:REM BLUE
550 IF A$ = " [CTRL] and [(8] "THEN
    POKE V+39,7:GOTO340:REM
    YELLOW
560 IF A$ = " [C=] and [!1] "THEN
    POKE V+39,8:GOTO340:REM
    ORANGE
570 IF A$ = " [C=] and ["2] "THEN
```

```
      POKE V+39,9:GOTO340:REM BROWN
580 IF A$ = "  and  "THEN
      POKE V+39,10:GOTO340:REM PINK
590 IF A$ = "  and  "THEN
      POKE V+39,11:GOTO340:REM DARK
      GREY
600 IF A$ = "  and  "THEN
      POKE V+39,12:GOTO340:REM
      MEDIUM GREY
610 IF A$ = "  and  "THEN
      POKE V+39,13:GOTO340:REM LIGHT
      GREEN
620 IF A$ = "  and  "THEN
      POKE V+39,14:GOTO340:REM LIGHT
      BLUE
630 IF A$ = "  and  "THEN
      POKE V+39,15:GOTO340:PRINT"
      THE FISH IS NOW INVISIBLE":GOTO340
640 IF A$="£" THEN POKE V+27,1—PEEK
      (V+27):GOTO 340:REM CHANGE
      PRIORITY
650 IF A$="F" THEN R=5:GOTO340:
      REM GO FAST
660 IF A$="S" THEN R=1:GOTO 340:
      REM GO SLOW
800 GOTO340
```


There are three main things to be learned about Sprites:

a) How to design them
b) How to put them into your program
c) How to control them on the screen.

THE DESIGN OF SPRITES

Sprite design is essentially an artistic matter. Every Sprite consists of a collection of bright points called *pixels*, drawn within a rectangle of 21 rows and 24 columns. A good place to start is to find some squared paper and draw a grid like this one:

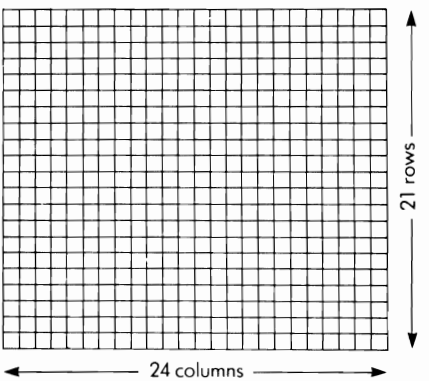


Figure 26.1

Now draw your Sprite on the grid, using pencil and *rubber*. Both 'solid' and 'line' drawings will work well. Eventually, you must end up with a *digitised* drawing — one where every square or pixel is either completely white or completely black. The result looks crude when you see it on the grid, but remember that the Sprite on the screen will be much smaller than your drawing, and the defects won't be visible.

The digitised drawing of the fish in the demonstration program looks like this:

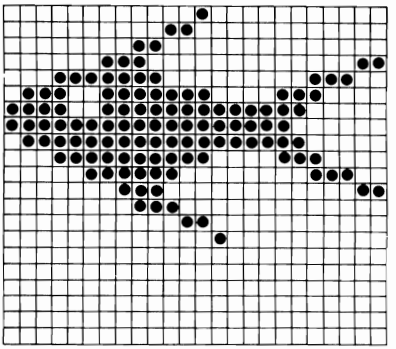


Figure 26.2

When your digitised drawing is ready, you must transcribe it into DATA statements which can be put into a BASIC program. You have to divide the pixels which make up the picture into 8-bit bytes, and then convert each byte into the corresponding decimal value. Here's how you might do it:

Take the first row of the picture, and split it into three groups of eight bits, like this:

00000000 00001000 00000000

(Use '1' to represent a filled pixel and 0 for an empty one.)

Now convert these three binary numbers into decimal, getting

0 8 0

(If necessary, see page 233 to remind yourself of the method.)

Now do the same for all the other rows of the picture. The second row will give (00000000, 00110000, 00000000) which converts to (0, 48, 0). The sixth row (taking one at random) will be (01110011, 11111000, 01110000) and gets translated to (115,248,112). You will end up with 63 decimal numbers which you can put into DATA statements like those in lines 30-90 of the demonstration program. Be sure to get them in the right order; left to right, and from the top down.

Of course the process you have just read about is unbearably tedious, especially if you are designing many different Sprites and are not too handy at converting binary numbers into their decimal equivalents. Not many people are. Life is made much simpler by using a suitable software design tool: a Sprite Editor Program.

EXPERIMENT 26·2

Load and run program MONSPR, which is a 'Sprite Editor'. It will let you draw a Sprite on the screen, move and turn it, or make it into a 'negative'. If you don't know what to do at any time, type 'H' and the program will display an explanation of all its facilities. Finally, when the picture of your Sprite is complete, type 'B' and the editor will do the binary-to-decimal conversions for you, displaying a set of DATA statements on the screen so that you can copy them into your own program.

When you've got the measure of the Sprite Editor, use it to design your own Sprite. Make a note of the DATA statements the Editor produces, and use them to modify the Sprite Demonstration Program, so that it displays *your* Sprite instead of the fish.

Experiment 26.2 Completed	

PUTTING SPRITES INTO A PROGRAM

Next we'll discuss putting Sprites into programs. Inside a program, the description of a Sprite takes up 63 consecutive bytes in the store. Each byte stands for eight pixels, and the bytes are stored from left to right and from the top down, in the same order as they were calculated from the digitised picture.

If a Sprite description could be placed into any 63 adjacent memory cells we could simply store it in a long string variable. Unfortunately this is not the case. For certain technical reasons a Sprite definition can only start at an address which is an exact multiple of 64. Furthermore, it will not easily go anywhere in the area between 4096 and 8191, or beyond 16383. If you look at the 64 memory map in Unit 23, you'll see that the areas where Sprite definitions are allowed are pretty much taken up with the Kernel, the Video Screen map, and the user's own program. We will just have to *make* space for the Sprites by pushing something else aside.

If your program uses only one or two Sprites, this is quite easy. It just happens that the block of 63 memory cells starting at 832 is dedicated to cassette tape transfers. So long as you have no intention of doing any tape transfers while your Sprite program is running, you can borrow these locations for your Sprite descriptions. 832 is exactly 13 times 64, so it is a suitable place to start. If you examine lines 120-140 of the demonstration program, you will see that they read the Sprite description data and POKE it into locations 832 onwards, one byte at a time.

How do you tell the 64 *where* you've put the Sprite description? (It won't always be 832.) For this job the system uses eight dedicated cells at the very end of the screen map: those with addresses 2040 to 2047. You'll remember that the video screen map has 1024 locations, but only 1000 of them are actually needed for the characters to be displayed on the screen. This is why cells 2040 to 2047 are actually available to help with Sprite Management.

At any moment, the machine can display up to *eight* different Sprites, which are numbered 0 to 7. Location 2040 is always connected with Sprite 0; location 2041 with Sprite number 1, and so on.

For any Sprite which is in use, the cell in the screen map holds a *pointer* to the corresponding Sprite description. The pointer is simply the address of the first byte of the description, *divided by 64*. For example, if the definition of Sprite 0 starts at address 832, we must arrange for cell 2040 to hold the number 13 (because 832 divided by 64 is exactly 13). Look at line 300 of the Demonstration program, which does exactly this job. It says,

```
300 POKE 2040,13
```

What would the statement have been if the program were using Sprite number 4, with the Sprite description at address 2112? See the bottom of the page for the answer.

Answer: 300 POKE 2044, 33

THE CONTROL OF SPRITES

The Sprites in a program are displayed and controlled by a special silicon chip called a Video Interface Controller (II). This is a marvellously powerful device, but it is also somewhat complicated. If all the controls were brought out to a panel, there would be something like 16 slide controls, 30 two-way switches, 15 multi-way switches, 20 indicator lamps and a couple of dials — enough, you might think, to fill the cockpit of an airliner. There is, however, a key difference: you don't need to understand all the controls before getting into the captain's seat and taking off!

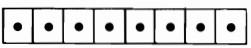
Like the sound generation chip (which you have already met) the Video Controller occupies a number of consecutive addresses. The first of these locations, which is called the 'base address', is at 53248. You might find this number easier to remember as '208 * 256'.

The Sprite Demonstration program uses a variable — V — to store the base address of the controller. This means that when writing the program we can refer to location 21 *in the controller* as "V+21" rather than "53269". This will help to make the program more readable and to avoid mistakes.

A full map of the Video Controller is shown in Figure 26.3. It can take care of up to eight Sprites at a time. You will see that some of the locations are shaded. Each one of these belongs to an *individual* Sprite so that, for example, location 7 is always used to control the vertical position of Sprite number 3. Other locations (the un-shaded ones) are shared amongst all the Sprites, each one being allocated one *bit* of the eight in the location. An example is location 23, which controls the vertical expansion of all eight Sprites (or at least of those currently on display).

Some of the locations in the chip are used for purposes completely unconnected with Sprites. For example, you already know about location 32 (frame colour) and 33 (background colour).

When all the Sprites share a location, the bits are arranged as follows:



Sprite number 7 6 5 4 3 2 1 0

The registers in the Video Controller are normally read and altered by PEEK and POKE commands. Suppose you are using only Sprite number 6, and you wish to expand it vertically. To do this you'll need to put the *binary* pattern 01000000 into location V+23. The BASIC command to do this is

```
POKE V+23,64
```

because 64 is the decimal equivalent of 01000000. This example presupposes, of course, that you have already set V to the base address of the controller.

The decimal numbers which correspond to the eight Sprites are:

Sprite number	7	6	5	4	3	2	1	0
Decimal equivalent for poking into shared locations	128	64	32	16	8	4	2	1

The appearance of a Sprite on the screen depends on many things. First, it must be switched on to be visible at all. Second, it has a position (so far across the screen, and so far down). Then it has a colour, horizontal and vertical sizes, and a priority which puts it either in front of, or behind other items displayed on the screen.

We'll explain all these controls, one by one. The examples assume that you are only displaying *one* Sprite at a time.

Location 21 of the controller is used to switch all the Sprites on and off. One bit is allocated to each Sprite, which only appears if that bit is a '1'. To turn Sprite 0 on, we put

```
POKE V+21,1
```

as in line 330 of the demonstration program. Note that '1' is the decimal code for Sprite 0. POKE V+21,0 would turn all the Sprites off (including Sprite 0).

The position of the Sprite (to be precise, the position of the top left-hand corner of the grid on which the Sprite was designed) is set by two coordinates: X (across the screen) and Y (down the screen).

The value of Y can vary between 0 and 255. To control the height of Sprite 0, you simply POKE the Y value into location number 1. For example, to put Sprite 0 100 units down, you could put

```
POKE V+1,100
```

Each Sprite has its own Y-register, so to position Sprite number 2 150 units down, you'd write

```
POKE V+5,150
```

Note that the allocation of location 5 to the vertical position of Sprite 2 is clearly indicated on the Controller map in Figure 26.3.

The X-position of a Sprite can take any value between 0 and 511. This is going to cause problems, since binary numbers in the range 0 to 511 need *nine* binary digits. This is one more than you can POKE into a single location.

The solution used in the Controller is to divide the X-value into two parts. The lower eight bits of the value are loaded into a dedicated register (just like the Y-value) but the uppermost digit is put into location 16, which is shared between all the Sprites and has a bit dedicated to each one.

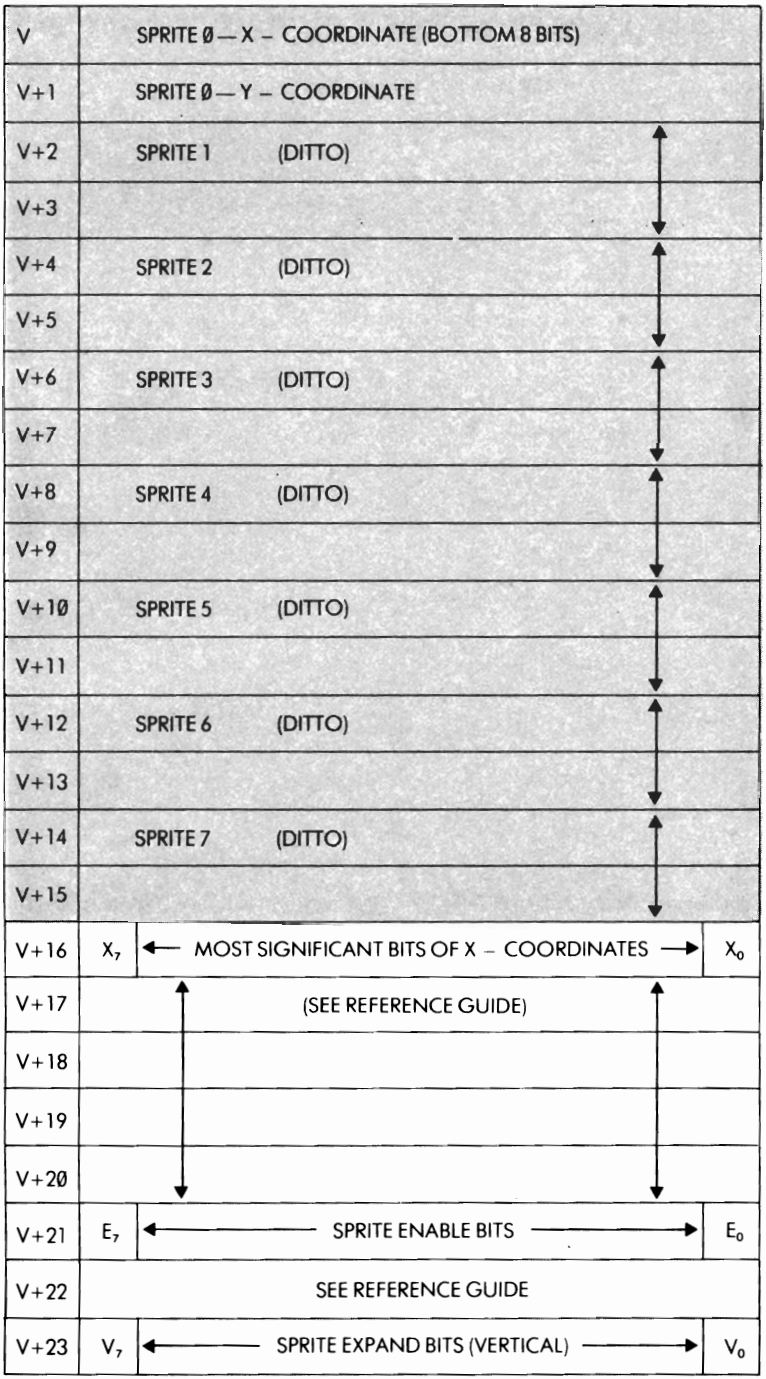

(53248)
V
SPRITE Ø — X – COORDINATE (BOTTOM 8 BITS)
V+1
SPRITE Ø — Y – COORDINATE
V+2
SPRITE 1 (DITTO)
V+3
V+4
SPRITE 2 (DITTO)
V+5
V+6
SPRITE 3 (DITTO)
V+7
V+8
SPRITE 4 (DITTO)
V+9
V+10
SPRITE 5 (DITTO)
V+11
V+12
SPRITE 6 (DITTO)
V+13
V+14
SPRITE 7 (DITTO)
V+15
V+16
X7
MOST SIGNIFICANT BITS OF X – COORDINATES
X0
V+17
(SEE REFERENCE GUIDE)
V+18
V+19
V+20
V+21
E7
SPRITE ENABLE BITS
E0
V+22
SEE REFERENCE GUIDE
V+23
V7
SPRITE EXPAND BITS (VERTICAL)
V0

Figure 26.3

V+24	MEMORY POINTERS		
V+25	↕ (SEE REFERENCE GUIDE) ↕		
V+26			
V+27	P_7	← SPRITE — DATA PRIORITY →	P_0
V+28	M_7	← SPRITE MULTICOLOUR BITS →	M_0
V+29	H_7	← SPRITE EXPAND BITS (HORIZONTAL) →	H_0
V+30	S_7	← SPRITE TO SPRITE COLLISIONS →	S_0
V+31	D_7	← SPRITE TO DATA COLLISIONS →	D_0
V+32	FRAME COLOUR		
V+33	BACKGROUND COLOUR		
V+34	↕ (SEE REFERENCE GUIDE) ↕		
V+35			
V+36			
V+37	FIRST COMMON COLOUR		
V+38	SECOND COMMON COLOUR		
V+39	SPRITE 0 COLOUR		
V+40	SPRITE 1 COLOUR		
V+41	SPRITE 2 COLOUR		
V+42	SPRITE 3 COLOUR		
V+43	SPRITE 4 COLOUR		
V+44	SPRITE 5 COLOUR		
V+45	SPRITE 6 COLOUR		
V+46	SPRITE 7 COLOUR		

Figure 26.3 (Part 2)

In BASIC, you can use the AND operation to separate out the lower 8 bits of the X-value, and make a test such as "IF X>=256 THEN " to see if the uppermost bit is a '1'. To put Sprite Ø into horizontal position X we would write:

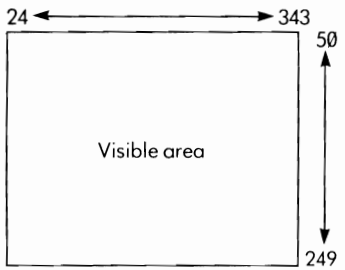
```
500 POKE V+0 , X AND 255
510 IF X>=256 THEN POKE V+16,1:
    GOTO 530
520 POKE V+16,0
530 REM X-POSITION NOW SET UP
```

These ideas can be generalised to any Sprite. For example, to put Sprite number N into position XN,YN (where N, XN and YN are variables) we can write:

```
600 POKE V+2*N , XN AND 255
620 IF XN>=256 THEN POKE V+16, 2↑N:
    GOTO 640
630 POKE V+16,0
640 POKE V+2*N + 1 , YN
```

This fragment of a program will not work if you are displaying more than one Sprite! The reason is explained later.

Not all the Sprite positions are actually visible on the screen. The boundaries for the top left corner of a Sprite are shown on the diagram below:



This diagram shows you that, for example, a Sprite with a Y-coordinate of 36 would be only partly visible, whilst one with an X-coordinate of 350 would not be seen at all.

The other controls are quite simple. The bits in location 23 look after the vertical size of the Sprites, and the ones in location 29 control their horizontal size. Thus the command

```
POKE V+29,1
```

will stretch Sprite number Ø to double its normal width. Similarly,

```
POKE V+29, 8 — PEEK(V+29)
```

will *change* the width of Sprite number 3 (broad to narrow, or narrow to broad). Can you see why?

The bits in location 27 control the priority of the Sprites: a 'Ø' in any bit puts the corresponding Sprite in front of the text on the screen, whilst a '1' puts it behind.

Finally, each Sprite has its own dedicated location for colour control. Register 39 belongs to Sprite Ø, register 40 to Sprite 1, and so on. To make a Sprite appear in a particular colour, you would put the corresponding code (for example, 2 for 'red') into the appropriate colour register.

EXPERIMENT 26·3

To check that you've understood all this new material, try your hand at modifying the Fish Demonstration Program so that it uses Sprite number 7 instead of Ø. The program should still run and produce an identical picture on the screen. You'll find the chart in Figure 26.3 useful, and you will not need to change the position of the Sprite Description in 832 onwards.

Experiment 26.3 Completed	

STUDY OF FOUR SPRITE PROGRAMS

Sprites which represent moving objects must be carefully controlled to give the appearance of smooth motion. The general method is to use a loop which corresponds to a fixed interval of time, and to show the Sprite at a slightly different position each time round. This gives the viewer the illusion of movement, just like an animated cartoon. To be effective, the time interval should not be longer than about 1/16 of a second, otherwise the Sprite will be seen to move in a series of jerks.

As it is going round the loop, the computer can either *calculate* the next position of the Sprite, or *look it up* in a table. The loop ends when some *boundary condition* occurs. This could be the Sprite hitting the edge of the screen or another object, or it could simply be that the loop has been obeyed some fixed number of times.

To demonstrate different ways of controlling Sprite motion, we give four short programs. These programs have no embellishments such as sound or attractive coloured displays; each one has been reduced to the minimum needed to illustrate the points being discussed.

The simplest kind of motion is movement in a straight line at unchanging speed. Load and run program LINEAR, and then have a look at the listing below:

```
  5 REM LINEAR
 10 REM PROGRAM TO SHOW MOTION
    OF BALL
 20 DATA0,0,0,0,0,0,0,112,0
 30 DATA3,254,0,15,255,128,31,255,192
 40 DATA63,255,224,63,255,224,127,255,
    240
 50 DATA127,255,240,127,255,240,63,255,
    224
 60 DATA63,255,224,31,255,192,15,255,128
 70 DATA3,254,0,0,112,0,0,0,0
 80 DATA0,0,0,0,0,0,0,0,0
100 REM SET UP SPRITE DESCRIPTION
110 FOR J=0TO62
120 READA: POKE 832+J,A
130 NEXT J
140 V=53248 :REM SET BASE ADDRESS
150 POKE 2040,13:REM SET POINTER TO
    DESCRIPTION
155 PRINT" [SHIFT] and [CLR HOME] ":REM
    CLEAR SCREEN
160 POKE V+33,1:REM SET
    BACKGROUND
170 POKE V+39,0:REM SET COLOUR
180 POKE V+23,1:POKEV+29,1:REM
    EXPAND
190 X=0:Y=0:REM SET STARTING
    POSITION
200 DX=2.3 :DY=1.2:REM SET
    HORIZONTAL AND VERTICAL SPEEDS
205 POKEV+21,1
210 POKEV+21,1 :REM ENABLE SPRITE
220 REM LOOP STARTS HERE
```

```
230 X=X+DX:Y=Y+DY :REM
    CALCULATE NEW POSITION
240 POKE V+1,Y
250 POKE V+0,X AND 255: REM PUT
    SPRITE THERE
260 IF X >=256 THEN POKE V+16,1:GOTO
    280
270 POKE V+16,0
280 IF Y < 225 AND X < 400 THEN230 :REM
    LOOP AROUND
290 GOTO190 :REM REPEAT MOVEMENT
```

The first part of the program sets up a Sprite description and initialises the Video Controller. The interesting part starts at line 190.

X and Y represent the position of the Sprite at any moment. We start at (0,0) which is off the screen.

DX and DY are the amounts by which X and Y *change* in every time interval. Thus, after one interval, Y will be 0+1.2 = 1.2; after two intervals, 2.4; and so on.

Once inside the loop, line 230 advances the current values of X and Y. Lines 240 to 270 are responsible for displaying the Sprite in its new position. The boundary condition is not reached until Y exceeds 225 or X exceeds 400, whichever comes first. While the current position of the Sprite is still within the screen the loop continues to go round and round.

EXPERIMENT 26·4

To get some feel for the working-of the program, stop it and alter the starting position and the values of DX and DY. You should be able to get the ball moving at different speeds and in different directions. To make the ball move to the left or upwards, try using negative values for DX and DY: but remember to alter the boundary condition and the starting place accordingly.

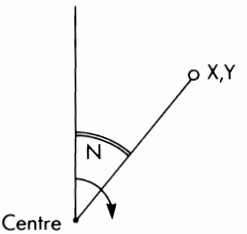
Experiment 26.4 Completed	

Program CIRCULAR1 moves the Sprite in a roughly circular path. To understand the details of the program, you'll need a little simple trigonometry. If this alarms you, skip to the next section.

Load program CIRCULAR1 and run it. You will see from the listing below that the method of control is quite different. The Sprite is constantly changing direction, and it never reaches the end of the screen!

```
  5 REM CIRCULAR1
 10 REM PROGRAM TO SHOW MOTION
    OF BALL
 20 DATA0,0,0,0,0,0,0,112,0
 30 DATA3,254,0,15,255,128,31,255,192
 40 DATA63,255,224,63,255,224,127,255,
    240
 50 DATA127,255,240,127,255,240,63,255,
    224
 60 DATA63,255,224,31,255,192,15,255,
    128
 70 DATA3,254,0,0,112,0,0,0,0
 80 DATA0,0,0,0,0,0,0,0,0
100 REM SET UP SPRITE DESCRIPTION
110 FOR J0TO62
120 READA: POKE 832+J,A
130 NEXTJ
140 V=53248 :REM SET BASE ADDRESS
150 POKE 2040,13:REM SET POINTER TO
    DESCRIPTION
155 PRINT " SHIFT and CLR HOME ": REM
    CLEAR SCREEN
160 POKE V+33,1:REM SET
    BACKGROUND
170 POKE V+39,0:REM SET COLOUR
180 POKE V+23,1:POKEV+29,1:REM
    EXPAND
190 FOR N=1 TO 100 STEP 0.15
205 POKEV+21,1
210 POKEV+21,1 :REM ENABLE SPRITE
220 REM LOOP STARTS HERE
230 X=150+ 40*SIN(N):Y=120-30
    *COS(N)
240 POKE V+1,Y
250 POKE V+0,X AND 255: REM PUT
    SPRITE THERE
260 IF X >=256 THEN POKE V+16,1:GOTO
    280
270 POKE V+16,0
280 NEXTN
290 GOTO190 :REM REPEAT MOVEMENT
```


Imagine the Sprite is whirling round your head on a piece of string. The string will move through the same small *angle* in every interval of time. In this program, the angular position of the string is represented by N, and the step size is 0.15 *radians* (about 7½ degrees). The position of the ball depends on the angle N according to the following diagram:



X = 150 +40*sin(N)
Y = 120 −40*cos (N)

Centre of circle at (150,120)
Radius of circle = 40

The 'boundary condition' in this case is simply that the ball has rotated through 100 radians — about 16 revolutions.

The snag with this program is that it is too slow to give a proper sense of motion. This is because the SIN and COS functions which are used inside the main control loop take a long time to work out.

```
  5 REM CIRCULAR2
 10 REM PROGRAM TO SHOW MOTION
    OF BALL
 20 DATA0,0,0,0,0,0,0,112,0
 30 DATA3,254,0,15,255,128,31,255,192
 40 DATA63,255,224,63,255,224,127,255,
    240
 50 DATA127,255,240,127,255,240,63,255,
    224
 60 DATA63,255,224,31,255,192,15,255,
    128
 70 DATA3,254,0,0,112,0,0,0,0
 80 DATA0,0,0,0,0,0,0,0,0
100 REM SET UP SPRITE DESCRIPTION
110 FOR J0TO62
120 READA: POKE 832+J,A
130 NEXTJ
140 V=53248 :REM SET BASE ADDRESS
150 POKE 2040,13:REM SET POINTER TO
    DESCRIPTION
155 PRINT " SHIFT and CLR HOME ": REM
    CLEAR SCREEN
160 POKE V+33,1:REM SET
    BACKGROUND
170 POKE V+39,0:REM SET COLOUR
180 POKE V+23,0:POKEV+29,0:REM
    EXPAND
190 DIM XP (80),YP(80)
```

```
200 FOR N=1TO80
210 T= 4.5*N*π/180
220 XP(N)=170+100*SIN(T):YP(N)=130
    -80*COS(T)
230 NEXTN
240 POKEV+21,1 :REM ENABLE SPRITE
250 FOR N=1 TO 80
260 REM LOOP STARTS HERE
270 X=XP(N):Y=YP(N)
280 POKE V+1,Y
290 POKE V+0,X AND 255: REM PUT
    SPRITE THERE
300 IF X>=256 THEN POKE V+16,1:GOTO
    320
310 POKE V+16,0
320 NEXTN
330 GOTO240 :REM REPEAT MOVEMENT
```

CIRCULAR2 illustrates a rather better method. Here 80 positions round the circle are calculated in advance and stored in arrays XP and YP. This is done in lines 190 to 230 of the program, which is displayed above. Then, when the Sprite is moving, its next position can simply be fetched from the table instead of being calculated afresh every time round the loop.

For circular motion, it is always worth while to calculate the positions in advance and store them in a table. Load and run CIRCULAR2, and note the improvement.

By using different mathematical functions to work out the table, you can get the Sprite to move in interesting trajectories. Here are some examples:

```
220 XP(N) = 150 +100 * SIN( 3*T) :
    YP(N) = 120 - 80*COS(T)
```

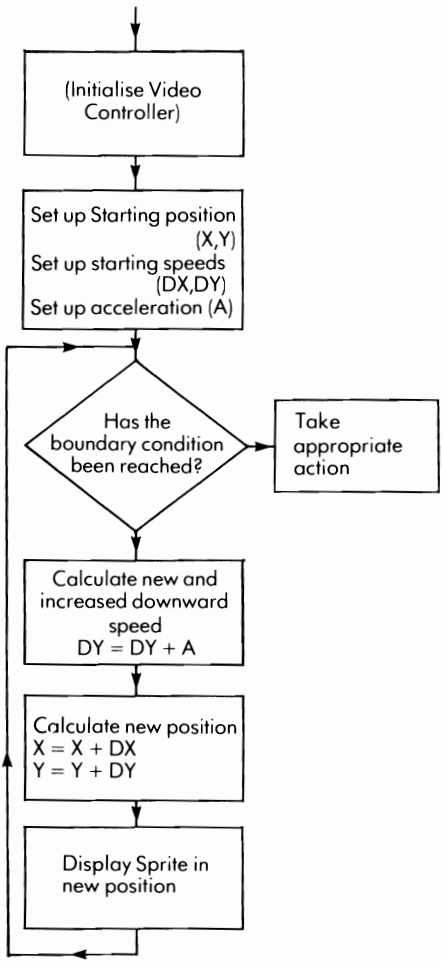
or

```
220 XP(N) = 150 + 80*SIN(T)+40*SIN
    (2*T)+20*SIN(3*T)
225 YP(N) = 120 - 60*COS(T)+30*COS
    (2*T)-15*COS(3*T)
```

An interesting location in the Video Controller is register 31. There is one bit for each of the eight Sprites. Whenever a moving Sprite collides with a stationary object on the screen (not another Sprite) the bit which belongs to the moving Sprite is set to '1'. You can use the register to detect balls bouncing against walls, bombs hitting the ground, or Tarzan being caught by the Cannibals. The technique is to use the register as a boundary condition, reading it just after you've moved the Sprite. The act of reading the register automatically resets it to zero (but not always immediately).

To complete this section we'll consider making Sprites move like falling objects. According to Newton the *horizontal* speed of a falling object stays the same; but the vertical speed downwards *increases* each second by a constant amount called the acceleration. The acceleration downwards is due to the force of gravity.

The core of a program to imitate a falling object would be like this:




```
  5 REM BOUNCING BALL
 10 REM PROGRAM TO SHOW MOTION
    OF BALL
 20 DATA0,0,0,0,0,0,0,112,0
 30 DATA3,254,0,15,255,128,31,255,192
 40 DATA63,255,224,63,255,224,127,255,
    240
 50 DATA127,255,240,127,255,240,63,255,
    224
 60 DATA63,255,224,31,255,192,15,255,
    128
 70 DATA3,254,0,0,112,0,0,0,0
 80 DATA0,0,0,0,0,0,0,0,0
100 REM SET UP SPRITE DESCRIPTION
110 FOR J=0TO62
120 READA: POKE 832+J,A
130 NEXTJ
```

```
140 V=53248 :REM SET BASE ADDRESS
150 POKE 2040,13:REM SET POINTER TO
    DESCRIPTION
160 POKE V+33,1:REM SET
    BACKGROUND
170 POKE V+39,0:REM SET COLOUR
180 POKE V+23,0:POKEV+29,0:REM DO
    NOT EXPAND
190 REM DRAW STAIRCASE
200 C$ = " [CTRL] and [!/1] [CTRL]
    and [RVS ON] [SPACE] 18 times [CTRL]
    and [RVS OFF] [SPACE] 7 times [CTRL]
    and [RVS ON] [SPACE] 19 times"
210 PRINT" [SHIFT] and [CLR HOME] [CRSR ↓]
    12 times"
220 FOR J=1TO13
230 PRINTLEFT$(C$,3*J+2)
240 NEXTJ
250 POKEV+21,1 :REM ENABLE SPRITE
260 X=20:Y=20:REM SET STARTING
    POSITION
270 DX=2.3:DY=0:A=1.4
280 REM LOOP STARTS HERE
285 IFPEEK(V+31)=1ANDDY>0THENDY
    =-0.6*DY:Y=Y+DY:POKEV+1,Y
290 X=X+DX:Y=Y+DY:DY=DY+A
300 POKE V+1,Y
310 POKE V+0,(X AND 255)
320 IF X>=256 THEN POKE V+16,1:GOTO
    340
330 POKE V+16,0
340 IF X < 450 AND Y+DY<255 THEN 285
350 GOTO 260
```

The principle is illustrated in program BOUNCE, which you should load and run. Here there are two boundary conditions. The first occurs when the ball hits a step, and is detected by the test in line 285. The condition (DY>0) is included to make sure that the ball is actually moving downwards. Register 31 is sometimes not reset immediately it has been read, and if this condition is omitted it can occasionally register a false 'hit' when the ball is on its way up.

The 'appropriate action' in this case is to reverse the vertical speed of the ball. If the ball were perfectly elastic, then the upward speed after hitting the step would be exactly the same as the downward speed before; we could put "DY = −DY". In practice, only a fraction of the speed is retained. "0.6" is about right for an ordinary rubber ball. A steel ball-bearing bouncing on a steel step might keep as much as 0.9 of its speed, whereas a squashy ball would only retain about a half.

EXPERIMENT 26·5

When you have studied the listing of the program, try altering some of the numbers, such as DX (the forward speed), A (the acceleration due to gravity) and the fraction of speed retained after the bounce. See if you can make the ball jump over the gap in the stairs.

Experiment 26.5 Completed	

EXPERIMENT 26·6

Try your hand at writing programs which use Sprites. Here are some suggestions:

a) a program which makes a Sprite move in an ever-decreasing spiral

b) a program which simulates a ball rolling about on a billiard table

c) a program which lets you steer a bob-sleigh along a Cresta run, to see how far you can get without hitting the walls.

Experiment 26.6 Completed	

MULTICOLOUR SPRITES

Next, we shall discuss another useful facility: one for displaying Sprites in more than one colour.

As you already know, a one-colour Sprite is made up of 21 rows of 24 pixels each. A multicolour Sprite also consists of 21 rows, but each row holds only 12 pixels. The pixels are twice as wide, so that the overall 'shape' of the Sprite is the same.

To make up for less detail, each pixel can be one of four different colours:

- It may be the *background* colour that is invisible.
- It may be a colour specially associated with the Sprite itself — the so called *primary* colour.
- It may be either of two *common* colours which are shared among all the Sprites on display at any one moment.

The actual colours displayed depend on the current setting of the Video Controller and may be changed from time to time.

To tell the controller which colour to use, each pixel in a multi-colour Sprite uses two binary digits. Each pair is as follows:

00 — Background colour
01 — First common colour
10 — Primary colour
11 — Second common colour

As you can see, each line of the Sprite has half the number of pixels, but each pixel needs double the amount of space — so the total amount of storage space needed for a multi-colour Sprite description is exactly the same as for a one-colour Sprite.

To get some idea of the appearance of a multi-colour Sprite, load and run program BUS. You can use this program as a guide when you come to use your own multi-colour Sprites.

```
  5 REM GLASGOW BUS
 10 REM PROGRAM TO SHOW MOVING
    MULTICOLOUR SPRITE
 20 REM DATA DESCRIBING GLASGOW
    BUS
 30 DATA0,0,0,85,85,84,65,4,4
 40 DATA65,4,4,85,85,84,85,170,84
 50 DATA255,255,252,195,12,48,195,12,48
 60 DATA255,255,255,255,255,255,251,
    255,251
 70 DATA8,0,8,0,0,0,0,0,0
 80 DATA0,0,0,0,0,0,0,0,0
 90 DATA0,0,0,0,0,0,0,0,0
100 REM SET UP SPRITE DESCRIPTION
110 FOR J=0TO62
120 READA: POKE 832+J,A
130 NEXTJ
140 V=53248 :REM SET BASE ADDRESS
```

```
150 POKE 2040,13:REM SET POINTER TO
    DESCRIPTION
160 POKE V+1,160 :REM SET
    (CONSTANT) HEIGHT OF SPRITE
170 POKE V+28,1:REM SET MULTI-
    COLOUR
180 POKE V+33,1:REM SET
    BACKGROUND
190 POKE V+37,7:REM SET FIRST
    COMMON COLOUR TO ORANGE
200 POKE V+38,5:REM SET SECOND
    COMMON COLOUR
210 POKE V+39,0:REM SET PRIMARY
    COLOUR
220 POKE V+23,1:POKEV+29,1:REM
    EXPAND
230 PRINT" SHIFT and CLR HOME CRSR
    16 times C and P 40 times"
240 POKE V,0:POKE V+16,0:POKEV+21,1
250 FOR X=0TO360
260 POKE V+0,XAND255
270 IF X>=256 THEN POKE V+16,1:GOTO
    290
280 POKE V+16,0
290 NEXT X
300 POKE V+21,0
310 GOTO 240
```


There are a few points which you should remember when using this facility:

1) The version of the Sprite Editor to use is called COSPRED.

2) To indicate that a given Sprite uses more than one colour, you must set the correct bit in Register 28 of the controller.

3) The codes for the common colour should be loaded into locations 37 and 38.

DISPLAYS WITH MANY SPRITES

The last topic in this Unit is about the problems of displaying more than one Sprite at a time. There are several major difficulties but there are good ways of overcoming them. The whole section assumes that you have set variable V to the base address of the Video Controller, at 208★256.

The first problem lies in changing individual bits in the Sprite control registers without affecting the others in the same location. Suppose you are using two Sprites — say 0 and 1. At some point in the program you want to make Sprite number 1 disappear. If you give the command

```
POKE V+21,0
```

then *both* Sprites will vanish — not what you want at all. Let's try

```
POKE V+21 , 1
```

This works (that is, it leaves Sprite 0 enabled) but you had to know *when you wrote the program* that Sprite 0 was to be showing at that time. In many cases Sprites are controlled by the users of the program, and you (the programmer) will not know in advance which Sprites are going to be on display at any one moment.

It turns out that the only reliable method is to PEEK the control register, change the bit you want, and POKE it back again. To change a bit to a '1' you OR the control byte with a number which has a bit at the right place. To change a bit to a 0, you AND the control word with a number given in the following table:

Sprite number	Word to enable OR (sets a 1)	Word to disable AND (sets a 0)
0	1	254
1	2	253
2	4	251
3	8	247
4	16	239
5	32	223
6	64	191
7	128	127

Here are some examples. To enable Sprite number 1 (without changing the state of Sprite 0) you would put

```
POKE V+21, PEEK(V+21) OR 2
```

To disable Sprite5 (supposing that you were using it) the right command would be

```
POKE V+21, PEEK(V+21) AND 223
```

If you are using more than one Sprite, the same method has to be used with all the control registers which hold bits in common, such as the Sprite expansion and Sprite collision registers. In particular, this is the only safe way to alter the contents of register 16, which holds the most significant digits of the X-coordinates of all the Sprites. Suppose that you want to move Sprite number 6 to a horizontal position of H, without affecting the positions of the other Sprites. You must put

```
POKE V+16, (PEEK(V+16) AND 191) OR
(INT(H/256)★64)
```

(Notice that the expression

```
INT(H/256)★64
```

is worth 64 if H is 256 or more, but 0 if H is less than 256.)

The second problem is to find a good place for your Sprite definitions. You'll remember that each definition is 64 bytes long, and that it must start on a 'block' boundary — that is, at an address which is an exact multiple of 64. Unfortunately the area from 832 onwards only has room for 3 Sprite

definitions, and if you try to store any more you'll run over into the screen area at 1024.

There are two ways to get round this problem. One method is to put your Sprite definitions 'high up' in the store (say at locations 15360 onwards) and hope that your program and data won't reach that far. If they do, then the program will simply crash without warning or explanation. The method works well for small programs, but is not recommended for serious work.

Another, much safer system is to use the Video Controller chip's facility to 'change blocks'. Normally the chip is constrained to fetch all its data — Sprite descriptions and the like — from the first 16K block of RAM in the 64 memory. However, it can be redirected to use the last 16K block instead. With a block change the Sprite definitions can now be stored in the RAM hidden 'behind' the Kernel ROM, as was explained in Unit 24. Unfortunately the block change has the unwanted side-effect of moving the screen map as well, so that 'ordinary' characters can no longer be displayed at the same time as the Sprites. The block change system itself is complicated and would take several pages to explain, but the instructions you need to write are quite simple and straightforward. They are as follows:

1. Store the Sprite definitions in addresses 58368 onwards. You can put them there without having to select the RAM, as the Kernel ROM is 'transparent' when you store information into the addresses it shares with the RAM underneath. You will have room for 112 different Sprite descriptions — enough for any program.

If your program uses — say — 12 Sprite definitions which are supplied as DATA statements, the instructions you need to set up the definitions are:

```
FOR J=0 TO 12*64-1:READ X:
POKE J+58368,X : NEXT J
```

This presupposes that the STATEMENTS for each Sprite contain 64 numbers — 63 to describe the Sprite itself and a zero to pad the length out to 64 bytes.

2. The 'block numbers' of these Sprites are '144' for the one which starts in 58368, '145' for the one in address 58432, and so on, up to '255' for the one which starts in 65472. The block change will move the screen map to addresses 57374 to 58367. Before you display any Sprites you must clear the new screen map to zeros, and put Sprite block pointers into the last 8 addresses. For example, if you want to use the 8 Sprites with block numbers 136 to 143, you would write,

```
FOR J=0TO 999 : POKE 57344+J,0: NEXT J
FOR J=0 TO 7: POKE 58360+J,144+J:
NEXT J
```

Of course you can redefine any of the Sprites at any time. If, for example, you want Sprite 1 to be the one in block 159, you would write

```
POKE 57361,159
```

When all the Sprite definitions and the new screen map have been set up, you 'switch them on' by the commands

```
POKE V+24,PEEK(V+24) OR 136
POKE 56578,PEEK(56578) OR 3
POKE 56576,PEEK(56576) AND 252
```

Your Sprites (up to 8 at a time) will now appear as soon as you enable them and put the correct positions into the X and Y registers of each one. On the other hand, the normal display system of the 64 is now disabled, and you will not be able to PRINT or POKE characters to the screen in the usual way. Error messages won't appear either! If you feel this is essential for your program, or if you would like to find out exactly how this system works, you are advised to read the COMMODORE 64 Programmers' Reference Manual, which contains a complete description of the mechanism.

At this point, you might find an illustration useful. Load and run program SOLAR, for which a listing is given below:

```
10 REM SOLAR
20 E=380 :L=27:REM SET NUMBER OF
   ORBITAL POINTS
30 ER=90:LR=25 :REM SET ORBIT SIZES
   OF EARTH AND MOON
40 DIM P2(E),Q2(E),P3(L),Q3(L)
50 PRINT" [SHIFT] and [CLR HOME]
   [CRSR][CRSR][CRSR][CRSR][CRSR][CRSR]
   CALCULATING ORBITS PLEASE WAIT"
60 PRINT:PRINT" ABOUT 45 SECONDS!"
240 REM SET UP SPRITE DEFINITIONS
250 FORJ=0TO6*64-1
260 READ X:POKE 58368+J,X
270 NEXT J
290 V=208*256
300 POKE V+32,0:POKE V+33,0 :REM
   SET SCREEN BLACK
320 FOR J=57344 TO 58343:POKE
   J,0:NEXT J:REM CLEAR NEW SCREEN
   MAP
325 FOR J=0 TO 5:POKE58360+J,144+J:
   NEXT J:REM SET SPRITE POINTERS
400 POKE V+41,7:POKE V+42,7:POKE
   V+43,7:POKE V+44,7 :REM SET SUN
   YELLOW
410 POKE V+40,5:POKE V+39,6:REM SET
   EARTH GREEN AND MOON BLUE
420 POKE V+4,160:POKE V+5,130:REM
   POKE COORDINATES OF SUN (4
   QUADRANTS)
430 POKE V+6,184:POKE V+7,130
440 POKE V+8 ,160:POKE V+9 ,149
450 POKE V+10,184:POKEV+11,149
```

```
460 PRINT" SHIFT and CLR HOME "
470 REM CALCULATE ORBITS
520 S=2★π/E: REM NEXT EARTH
530 FOR J=1 TO E
540 P2(J)=INT(1.2★ER★SIN(S★J)+184):
    Q2(J)=INT(ER★COS(S★J)+149)
550 NEXT J
560 S=2★π/L: REM NEXT THE MOON
570 FOR J=1 TO L
580 P3(J)=INT(1.2★LR★SIN(S★J)):
    Q3(J)=INT(LR★COS(S★J))
590 NEXT J
680 POKEV+24,136 :REM MOVE SCREEN
    AND CHARACTER TABLE
690 POKE 56578,PEEK(56578)OR3:POKE
    56576,PEEK(56576) AND 252 :REM
    SELECT BANK 3
700 EM=1:LM=1
705 POKE V+21,63 :REM ENABLE SPRITES
710 FOR Q=1 TO 100000
740 POKEV+2,P2(EM) AND 255:POKE
    V+3,Q2(EM)
741 POKE V+16,(PEEK(V+16)AND253)+
    INT(P2(EM)/256)★2
745 PP=P2(EM)+P3(LM):QQ=Q2(EM)
    +Q3(LM)
750 POKEV+0,PP AND 255:POKE
    V+1,QQ AND 255
751 POKE V+16,(PEEK(V+16)AND254)+
    INT(PP/256)
760 EM=EM+1:IFEM>ETHEN EM=1
770 LM=LM+1:IF LM>L THEN LM=1
800 NEXT Q
1400 REM SPRITE DEFINITIONS
1450 REM MIDDLE-SIZED PLANET
1460 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
1470 DATA0,124,0,1,255,0,3,255,128,3,255,
    128,7,255,192,7,255,192,3,255,128
1480 DATA1,255,0,0,124,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0
1500 REM BIG PLANET
1510 DATA0,0,0,0,24,0,1,255,128,7,255,224,
    31,255,248,63,255,252
1520 DATA127,255,254,127,255,254,255,
    255,255,255,255,255,255,255,255,255,
    255,255
1530 DATA127,255,254,127,255,254,63,
    255,252,31,255,248,7,255,224,1,255,
    128
1540 DATA0,24,0,0,0,0,0,0,0,0
1550 REM SUN (1ST QUADRANT)
1560 DATA0,0,3,0,0,127,0,3,255,0,31,255,
    0,127,255,1,255,255,3,255,255,7,255,
    255
1570 DATA15,255,255,31,255,255,31,255,
    255,63,255,255,63,255,255,127,255,
    255
1580 DATA127,255,255,255,255,255,255,
    255,255,255,255,255,255,255,255,255
1590 DATA0,0,0,0,0,0,0,0
1600 REM SUN (2ND QUADRANT)
1610 DATA192,0,0,254,0,0,255,192,0,255,
    248,0,255,254,0,255,255,128,255,255,
    192
1620 DATA255,255,224,255,255,240,255,
    255,248,255,255,248,255,255,252,
    255,255,252
1630 DATA255,255,254,255,255,254,255,
    255,255,255,255,255,255,255,255,
    255,255,255
1640 DATA0,0,0,0,0,0,0,0
1650 REM SUN (3RD QUADRANT)
1660 DATA255,255,255,255,255,255,255,
    255,255,255,255,255,255,255,255,255
1670 DATA127,255,255,127,255,255,63,
    255,255,63,255,255,31,255,255,31,
    255,255
1680 DATA15,255,255,7,255,255,3,255,
    255,1,255,255,0,127,255,0,31,255,
    0,3,255
1690 DATA0,0,127,0,0,3,0,0,0,0,0,0,0,0
1700 REM SUN (4TH QUADRANT)
1710 DATA255,255,255,255,255,255,255,
    255,255,255,255,255,255,255,255,255
1720 DATA255,255,254,255,255,254,255,
    255,252,255,255,252,255,255,248,
    255,255,248
1730 DATA255,255,240,255,255,224,255,
    255,192,255,255,128,255,254,0,255,
    248,0
1740 DATA255,192,0,254,0,0,192,0,0,0,0,0,
    0,0,0,0,0
```


SOLAR uses 6 Sprites: one for the earth, one for the moon and four for the four quarters of the sun, which is too big to be shown by one Sprite alone. The definitions of the Sprites are read from DATA statements and stored by the code in lines 240 to 270. The new screen map is cleared and set up with Sprite pointers in lines 320 and 325. Lines 470 to 590 work out tables to hold the orbits of the earth and the moon, so that the program will not have to do the complex calculations repeatedly when displaying the moving picture.

The new bank of store is selected by command 690, and the new screen map is set up by line 680. Finally, the planetary display is run by lines 710 to 800. The complex entries in 741 and 751 are needed to alter individual bits in register 16.

The third problem in displaying Sprites is the matter of speed. In SOLAR, the program you have just looked at, the heavenly bodies move only just quickly and smoothly enough to give a reasonable impression of continuous motion. If you have more than two moving Sprites, and you attempt to control them with a BASIC program, the display slows down drastically; the Sprites creep around in slow motion, and your program is more likely to disappoint than to impress. To see the problem for yourself, load and run program GROTTY, in which the number of moving objects have been increased from 2 to 4.

The best way to make your Sprites move fast is to use a subroutine written in machine code. Machine code is the computer's own language, and since no translation is needed it runs very much faster than BASIC. Machine code is complex and difficult to understand, and would need a

whole book to explain it properly; so here we'll just give you a suitable subroutine and the instructions you need to use it.

```
 10 DIM SP(10)
100 REM MACHINE CODE ROUTINE TO
    SERVICE SPRITES
110 REM IT UPDATES THEIR POSITION
    EVERY INTERRUPT
120 DATA165,47,133,251,165,48,133,252,
    160,0,177,251,201,83,208,8,200,177,
    251
130 DATA201,80,240,38,136,200,200,177,
    251,133,253,200,157,251,133,254,24,
    165,251
140 DATA101,253,133,251,165,252,101,
    254,133,252,197,50,208,212,165,251
150 DATA197,49,208,206,76,49,234
160 DATA152,24,105,6,168,162,0,200,200,
    24,177,251,41,1,240,1,56,102,255,200
170 DATA177,251,157,0,208,200,177,251,
    157,1,208,232,232,200,224,16,208,225
180 DATA165,255,141,16,208
182 DATA162,0,200,200,177,251,240,29,
    169,0,157,4,212,177,251,157,4,212,169,
    0
184 DATA145,251,200,177,251,157,1,212,
    200,177,251,157,0,212,76,208,3
186 DATA200,200,200,138,24,105,7,170,
    224,21,208,209,76,49,234
190 REM INSERT WEDGE (CODE AT 988)
200 DATA120,169,64,141,20,3,169,3,141,
    21,3,88,96
210 FORJ=832 TO 1000
220 READA:POKEJ,A:NEXTJ
230 SYS(988):A=2↑31:B=2↑8
```

The machine code routine in SPRMAC looks after the *position* of your Sprites, but does not concern itself with their colour or size. Since it is only the position which changes rapidly in a moving picture, this is just what is wanted.

To use the routine, make sure that it is included in your program. You could list it and copy it by hand, but we don't advise this method because even the smallest error would prevent the routine from working properly. It is far better to load the routine from cassette or diskette and build your program round it.

The routine should come at the beginning of your program, so that the first command to be obeyed are those in lines 210 to 230. These commands read the coded routine from the data statements and put it away in locations 832 to 1000, so those addresses cannot be used to hold Sprites!

The routine uses variables A and B, and array SP for special purposes. Your program must not use these variables except as described below. Once your program has obeyed the commands in lines 210 to 230, the rest is easy. To move a Sprite to a given position, all you need do is to assign a single value to one of the elements of SP: SP(0) for Sprite 0, SP(1) for Sprite 1, and so on. The value should be A + B times the X-coordinate + the Y-coordinate. For example, to put Sprite 4 at position (310,199) you write

```
SP(4) = A + B * 310 + 199
```

It is important that the X-coordinate be a whole number, so to be safe you would sometimes write

```
SP(2) = A + B*INT(X) + Y
```

Load and run program PLANETS, which is the same as GROTTY except that all the Sprite movements are looked after by the machine code routine. You'll see that the illusion of smooth motion has been restored. The main differences between GROTTY and PLANETS are as follows:

1. PLANETS includes the machine code routine near the beginning.

2. The POKEs which move Sprites around in GROTTY are replaced by assignments to elements of SP. Thus the command which positions the first quarter of the sun is changed from

```
POKE V+6,160: POKE V+7,130
```

to

```
SP(3) = A + B * 160 + 130
```

and the commands which place the comet are altered from

```
POKE V+14, P4(CM) AND 255
POKE V+15, Q4(CM)
POKE V+16, (PEEK(V+16) AND 127) +
INT(P4(CM)/256)*128
```

to

```
SP(7) = A+B*P4(CM)+Q4(CM)
```

The same machine code routine is also useful in playing music. This will be discussed further in Appendix A.

This Unit has given you only the briefest introduction to the use of Sprites on the 64. The Video Controller chip has far more potential than we have been able to describe, but a knowledge of machine code is needed to get the best performance out of it. One day I hope to write a book about it!

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 manyother 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 COMMODORE 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.

We end with ten 'precepts' of good programming. Oscar Wilde once said:

"I always pass on good advice; 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. If 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, 1983

APPENDICES

APPENDIX A

The COMMODORE 64 has a built-in sound synthesizer which can be used as a versatile and unusual musical instrument. It takes a lot of care and planning to get the best performance out of the machine, and you'll need to use art as well as science. The final test of a music program is not "Is it correct?", but "Does it sound good?". This is ultimately a matter of personal taste.

This Appendix is only an introduction to making music on the 64. The sound synthesizer chip is so flexible, and has so many different facilities, that it would take a whole book to describe it in full and explain how to use it to the best effect. One day such a book may come to be written; but in the meantime the best source of further information is the Commodore 64 Programmer's Reference Guide.

A vital part of music making is the production of individual notes. This topic was discussed in Unit 13. We strongly suggest that you re-read that unit and try out program ESG again before continuing with the Appendix. In particular, you should be comfortable with the notion of pitch, duration and timbre, and with the idea of the envelope of a note.

1. PLAYING TUNES

A tune is a sequence of notes, each with its own pitch and duration. The timbre and envelope are usually common to all the notes, since they would normally be played on the same instrument.

We already know how to make the machine play notes one at a time. In principle, to play a tune all we need to do is to supply the 64 with the particulars of each note and let it play them one after the other. Only certain details remain to be settled:

In Unit 13 we explained how to vary the pitch of a note by POKing numbers into addresses 54272 and 54273. The pitch depended on the frequency of the note, or the number of vibrations per second, and the correct sequence of commands, for a frequency F, was:

```
FF = 16*F
POKE 54273, FF/256
POKE 54272, (FF-32768) AND 255
```

To play a tune, we must know the frequency of every note. The following chart will help:

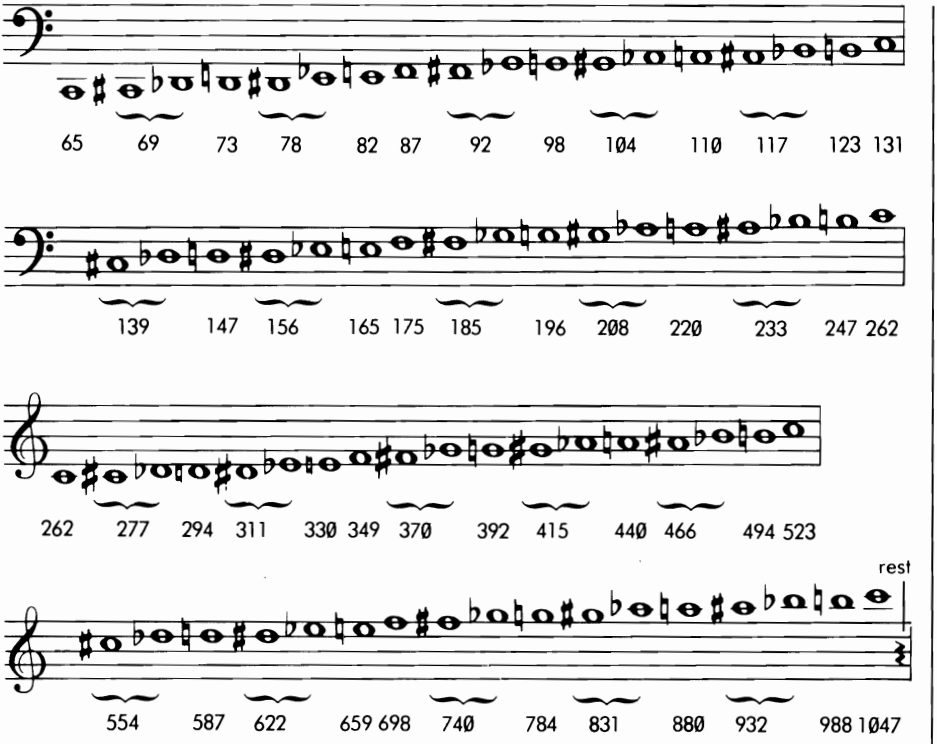


Figure A1

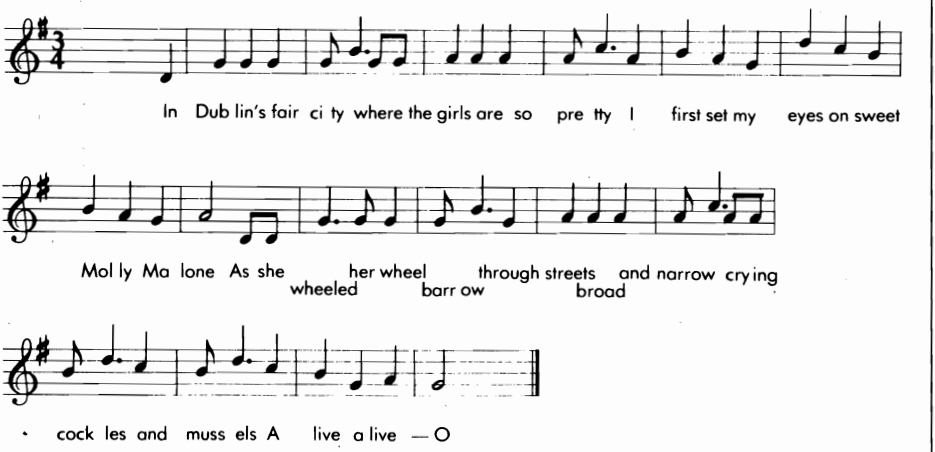


Figure A2

Next, we must be able to specify the length of each note. This is most easily done in beats, where a beat is the length of the shortest note in the piece.

Here is a program to play the first phrase of the tune in Figure A2. Each DATA statement (except the last) describes one note: the two numbers represent the frequency and the length (in quavers or half-notes). For example, the first note is a crotchet or whole note on D. Its frequency (taken from Figure A1) is 287, and its length is two half-notes. This gives

```
DATA 287 , 2
```

The last DATA statement marks the end of the tune by using the special frequency value zero.


```
10 REM IN DUBLIN'S FAIR CITY
20 VV=212*256 :REM SET BASE
   ADDRESS OF SOUND CHIP
30 POKE VV+24,15 :REM SET VOLUME
40 POKE VV+5, 9+16*1 :REM SET
   ATTACK AND DECAY
50 POKE VV+6,0:REM SET RELEASE
   VALUE
60 REM MAIN LOOP STARTS HERE
70 READ F,N : REM READ DETAILS OF
   NEXT NOTE
80 IF F=0 THEN POKE VV+24,0:STOP:
   REM 0 MEANS "END OF TUNE"
90 FF=16*F : REM SET FREQUENCY OF
   NOTE
100 POKE VV+1,FF/256
110 POKE VV,(FF-32768)AND 255
120 T=TI+15*N :REM SET ALARM FOR
   DURATION
130 POKE VV+4,17 :REM START NOTE
140 IF TI < T THEN 140 :REM WAIT FOR
   ALARM
150 POKE VV+4,0 :REM STOP NOTE
160 GOTO 60
200 DATA 287,2:REM IN
210 DATA 384.2:REM DUB-
220 DATA 384,2:REM LIN'S
230 DATA 384,2:REM FAIR
240 DATA 384,1:REM CI-
250 DATA483,3 :REM TY
260 DATA384,1 :REM WHERE
270 DATA384,1 :REM THE
280 DATA431,2 :REM GIRLS
290 DATA431,2 :REM ARE
300 DATA431,2 :REM SO
310 DATA431,1 :REM PRET-
320 DATA512,3 :REM TY
1000 DATA 0,0, :REM END OF TUNE
```

The program begins by setting variable VV to the 'base address' of the sound chip. This is at 54272, or in a form which some people might find easier to remember, 212*256. All the registers in the sound chip can now be referred to as offsets from the base address instead of using their absolute addresses. This makes the program shorter and easier to understand as well as reducing the danger of errors.

The control registers for the voice we are using are all close together, as follows:

Absolute location	Relative location	Purpose
54272	(VV+) 0	Pitch control
54273	(VV+) 1	Pitch control
54274	(VV+) 2	Pulse width control
54275	(VV+) 3	Pulse width control
54276	(VV+) 4	Waveform selection and envelop start/stop
54277	(VV+) 5	Attack and Decay control
54278	(VV+) 6	Sustain and release control

The volume control register is in address 54296, or (in relative terms), VV+24.

When the base address has been set in VV, the program sets the volume to maximum and selects attack and decay values to give a gently decaying note, rather like an Irish harp. Sustain and release values are not needed, so the register at VV+6 is set to zero. These settings will hold for the whole of the tune.

The main loop of the program, which is obeyed once for every note in the tune, starts at line 60. The machine reads the frequency and duration values for the next note. If the frequency is 0, this marks the end of the tune and the program switches off the sound and stops. Otherwise it sets up the pitch registers (in lines 90-110) and sets an 'alarm' to time the note. Each beat has been set to 15 jiffies, and the variable T is set to the time when the next note is due to end. For example, if the note is supposed to be three beats long, T is set to the current time plus 3*15 or 45 jiffies.

Line 130 starts the note, and then line 140 makes the machine wait until the internal timer TI catches up with the alarm time in T. When this happens, the machine switches off the sound (by POKing 0 into the register at VV+4) and jumps back to play the next note.

EXPERIMENT A3·1

You'll find a copy of this program on your cassette tape or diskette under the title DUBLIN. Load and run it. Then try a number of experiments:

First, extend the program so that it plays the whole tune (not just the first line).

Next, change the speed by altering the '15' in line 12Ø to some other number; for example, '3Ø' will make the tune twice as slow.

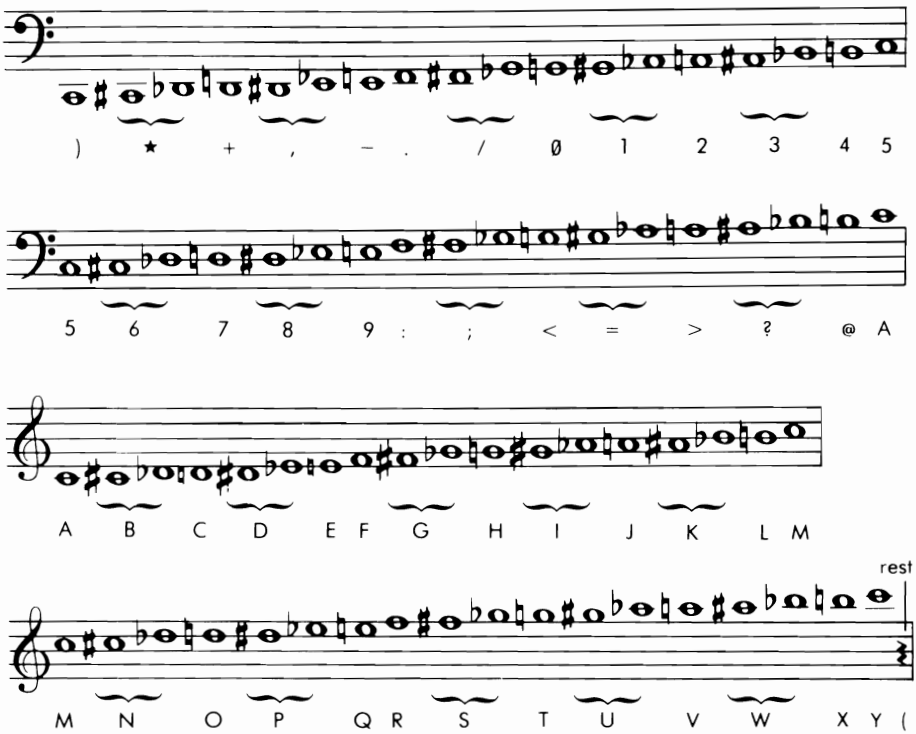
Finally, try changing the timbre by adjusting lines 4Ø, 5Ø and 13Ø. For instance, the following gives a sound like a trumpet:

```
4Ø POKE VV+2,2ØØ:POKE VV+3,Ø:POKE
    VV+5,16
5Ø POKE VV+6,24Ø
...
13Ø POKE VV+4,65
```

Experiment A3.1 Completed	

Experiment A3.1 will have convinced you that entering music into the 64 is very hard work and prone to errors. Another drawback is that each DATA statement takes up 16 bytes, and in a very large program you may not be able to spare the memory space for a long tune.

Pitch:



Duration

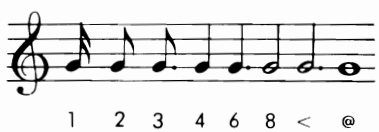


Figure A3

To make matters a little easier, we'll use a special notation which represents each note as just two characters: one for the pitch and one for the duration. The code is set out in Figure A3. It was invented specially for this book and is quite unlike any other code; in particular, it isn't the same as the code used in other Commodore products such as the VIC Super-expander. Nevertheless, you'll 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 63,64,65,66 and 67. This makes it easy to look up the right frequency from a suitable table. A rest (no sound at all) is indicated by the character "(" (left parenthesis).

The length of the notes are measured in semi-quavers or quarter-notes. Where a note is more than 9 semi-quavers 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 part of a tune you put down two strings. One string has all the pitch characters, and the other has 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:

```
DATA "CHHHHLHHJJJJMJLJHOMLLJHJ"
DATA "4444262244426444444444448"
DATA "CCHHHHHLHJJJJMJJLOMLOMLHJ
     H"
DATA "224442644442622264264444 8"
DATA "Z"
```

Another example, which is written in the bass clef and uses some rests, is shown below. *(from "Elijah" by F. Mendelssohn)*

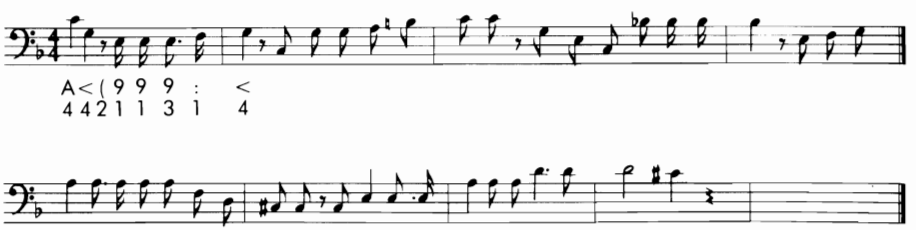


Figure A4


```
DATA "A<(999:<(5<<> AA(,95????(9:<"
DATA "4421131422222222222221146222"
DATA ">>>>>:766(6999>>>CCCB"
DATA "4312222222224314226284"
DATA "Z"
```

Here is a program which plays tunes written in this notation.

```
 10 REM TUNE PLAYER
 20 DIM N(60) :REM FREQUENCY TABLE
 30 FOR J=1 TO 60 :REM SET FREQUENCY
    TABLE
 40 N(J) = 64*1.059463 ↑ J
 50 NEXT J
 55 R=6
 60 VV=212*256:REM SET BASE ADDRESS
 70 POKE VV+24,15 :REM SET VOLUME
 80 POKE VV+5,9+16*1 :REM SET
    ATTACK AND DECAY
 90 POKE VV+6,0 :REM SET RELEASE
    VALUE
100 REM MAIN LOOP STARTS HERE
110 READ X$
120 IF X$="Z" THEN POKE VV+24,0:STOP
    :REM STOP IF END OF TUNE
130 READ Y$
140 FOR J=1 TO LEN(X$):REM PLAY EACH
    NOTE IN STRING PAIR
150 A=ASC(MID$(X$,J,1)):B=ASC(MID$
    (Y$,J,1))
160 FF=16*N(A-40)
170 POKE VV+1,FF/256
180 POKE VV,(FF-32768) AND 255
190 T=TI+R*(B-48):REM SET ALARM IN T
200 POKE VV+4,17 :REM START NOTE
210 IF TI<T THEN 210 :REM WAIT FOR
    NOTE TO END
220 POKE VV+4,0 :REM STOP NOTE
230 NEXT J
240 GOTO100
400 DATA"CHHHHLHHJJJJMJLJHOMLL
    JHJ"
410 DATA"4444262244426444444444448"
420 DATA"CCHHHHLHHJJJJMJJLOM
    LOMLHJH"
430 DATA"224442622444262226426462 48"
440 DATA"Z"
```

The program is basically similar to the first one in this section, but there are some important differences.

First, the program works out its own table of frequencies. The table is kept in array N, and the values are worked out in lines 30 to 50. The mathematical formula in line 40 is the one which relates pitch to frequency, and matters are arranged so that the frequency of middle C is put into N(25). Each element in the array corresponds to a different note on the piano, so that, for example, N(26) holds the frequency of C # (or Db), whilst N(44) holds the G an octave above middle C. Bass C is in N(1), and top C in N(49). N(40) holds 0, a 'frequency' which serves to play a rest.

Next, the program sets up the base address and the various control registers in a straightforward way. The main loop starts with line 110, which reads the 'pitch' string into X$. If a "Z" is found the program ends since a "Z" is used to

mark the end of the tune. Otherwise the program reads the 'duration' string into Y$, and enters the inner loop at 140. This loop is obeyed once for each note in the strings. Variable J is used to 'march' down both the strings at the same time, extracting another pair of characters each time round. The characters are converted into their ASCII codes and placed in variables A and B. Then A is used to index the frequency table, and B is used to control the length of the note. The offset (40) is needed in line 160 because the ASCII code for "A" (which stands for Middle C) is 65, but the frequency for Middle C is stored in N(25). 65−25=40. The offset 48 in line 190 is needed for a similar reason.

When the length of a note is chosen, the time is multiplied by variable R. The value of R is set in line 55, and different values will make the tune speed up or slow down accordingly.

This program can readily be adapted to play any tune you like.

2. HARMONY

The COMMODORE 64 has three distinct and separate 'voices', of which we have so far used only one. The control registers for voice 2 are in (relative) positions 7 to 13 on the sound chip, and the ones for voice 3 are in positions 14 to 20. They are used in exactly the same way as the voice 1 controls.

Since the three voices can all run at the same time, the 64 can play chords, and music in up to three separate parts. Furthermore, the parts can be given varying timbres so that the machine sounds like an ensemble of different instruments.

Unfortunately, when you attempt to write a program which implements these ideas, the results are very discouraging; the notes don't sound at the right time and the performance reminds you of a group of poor amateur musicians. The reason for this trouble lies in the speed at which BASIC is interpreted by the 64. Every note takes several commands to start, and since they can only be started one at a time, it becomes noticeable that notes which are meant to start together actually occur one after the other.

If you keep strictly to BASIC, there is no cure for this problem. The only way to solve the difficulty is to give the computer part of the program in Machine Code, the language which the processor can obey directly without interpretation. Machine code runs about 1000 times faster than BASIC, but it is difficult to write, and is not recommended for any but the most critical applications (such as this one).

The program given below includes a section of machine code (in lines 10 to 100) which allows all the notes in a 3-part piece to be played together. You can load the program from your cassette or diskette (it is called "SHEBA") and try it out.

If you understand the musical notation described in the previous section, you can easily adapt the program to play any music you like. You should change the DATA statements from line 1000 onwards, putting in the code for your music.

The detailed aspects of this program you may want to alter are as follows:

a) The speed of performance is governed by the *second* number in DATA statement 310. The larger the number, the slower the music.

b) The type of waveform used for each of the three voices is controlled by the *first* number in DATA statements 320, 330 and 340, respectively. In the "Arrival of the Queen of Sheba" the top voice uses a pulsed waveform which gives a 'twangy' sound, whilst the two lower voices use triangular waveforms with much more gentle characteristics. The overall ensemble is meant to resemble a two-manual harpsichord.

c) The other sound registers are set up in 500 to 565, and you can alter these commands to produce the kind of sound you like best.

d) The music itself is given in the DATA statements from line 1000 on. The statements come in groups of six, each group describing about four bars of music. The first two strings give the pitch and duration of the notes in the first voice; strings three and four apply to voice 2, and the last two strings describe voice number 3. The first 8 bars of the piece are written out in full so that you can study the notation.

```
1 REM QUEEN OF SHEBA
2 REM BY G F HANDEL (ARRANGED
  FOR COMPUTER BY A J T COLIN)
3 POKE 53280,0:POKE 53281,1
4 PRINT " SHIFT and CLR HOME CRSR
  3 times CRSR 4 times CTRL and
  ! 1 THE ARRIVAL OF"
5 PRINT " CTRL and ! 1 CRSR
  3 times CRSR 4 times SPACE
  10 times THE"
6 CTRL and ! 1 CRSR 3 times
  CRSR 4 times CTRL and # 3
  QUEEN OF SHEBA"
7 PRINT " CTRL and ! 1 CRSR
  6 times CRSR 3 times BY G F HANDEL"
10 REM SHEBA
20 DATAA5,01,29,FE,85,01,AD,00,A0,
   D0,09,A5,01,09,01,85,01,4C,31,EA,
   A2,00
30 DATABD,82,A0,F0,6D,FE,87,A0,D0,
   68,FE,88,A0,D0,63,BD,83,A0,85,FE,
   BD,84,A0
```

```
 40 DATA85,FF,A0,00,B1,FE,0A,A8,B9,
    02,A0,9D,00,D4
 50 DATAB9,03,A0,9D,01,D4,BD,85,A0,
    85,FE,BD,86,A0,85,FF,A0,00,B1,FE,
    D0,06
 60 DATA9D,82,A0,4C,C8,03,A8,BD,87,
    A0,38,ED,01,A0,9D,87,A0,BD,88,A0,
    E9,00
 70 DATA9D,88,A0,88,D0,EB,FE,83,A0,
    D0,03,FE,84,A0,FE,85,A0,D0,03,FE,
    86,A0,A9,00
 80 DATA9D,04,D4,BD,82,A0,9D,04,D4,
    8A,18,69,07,AA,C9,15,D0,03,4C,4B,
    03,4C,56,03
 90 REM HERE COMES WEDGE
    INSERTION CODE
100 DATA78,A9,40,8D,14,03,A9,03,8D,
    15,03,58,60
200 FOR J=832 TO 995
210 READ X$:L=ASC(LEFT$(X$,1)):R=ASC
    (RIGHT$(X$,1))
220 IF L >= 65 AND L <= 70 THEN Y=16*
    (L-55):GOTO 240
230 Y=16*(L-48)
240 IF R >= 65 AND R <= 70 THEN Y=Y+
    R-55:GOTO 260
250 Y=Y+R-48
260 POKE J,Y
280 NEXT J
300 REM SET UP CONTROL AREA
310 DATA 0, 8
320 DATA 65,151,160,103,168,255,255
330 DATA 17,55,176,7,184,255,255
340 DATA 17,215,191,167,199,255,255
350 READ A,B
360 POKE 40960,A:POKE 40961,B
370 F=1100:Q=2↑(1/12)
375 POKE 40962,0:POKE40963,0
380 FOR J=1TO63
390 POKE 40962+2*J,(F-32768)AND 255
400 POKE 40963+2*J,INT(F/256)
410 F=F*Q
420 NEXTJ
430 FOR J=0 TO 20: READ A: POKE
    41090+J,A:NEXTJ
500 REM SET UP VOICES
510 VV=212*256
520 FOR J=0 TO 24:POKE VV+J,0
530 POKE VV+5,25:POKE VV+12,25:
    POKE VV+19,25
560 POKE VV+24,79:POKE VV+2,200:
    POKE VV+9,60 :POKE VV+16,60
565 POKE VV+22,50 :POKE VV+23,0
570 IFP=0THEN600
580 FORJ=1TOP:READ A$,B$,C$,D$,E$,
    F$:NEXT
600 REM SET UP CONTROL AREAS
610 FOR J=0 TO 2
620 READ X$,Y$
621 IF X$="Z" THEN 750
630 Z=LEN(X$)
640 FOR Q=1 TO Z
650 N=ASC(MID$(X$,Q,1))-40: POKE
    41111+J*4000 + P(J),N
660 N=ASC(MID$(Y$,Q,1))-48: POKE
    43111+J*4000 + P(J),N
670 P(J)=P(J)+1
680 NEXT Q
690 NEXT J
700 POKE 40960,1
705 SYS(983)
710 GOTO 600
750 REM PLANT END OF TUNE
760 FOR J=0 TO 2
770 POKE 41111+J*4000+P(J),0
780 POKE 43111+J+4000+P(J),0
790 NEXT J
800 GOTO800
1000 DATA"WRORWRORWRORWROR
     MWVTRPOMOKMOPRTVWRKRWR
     ORWQMQWTMTVTRTMQRMOMO
     MO"
1010 DATA"1111111111111111111111111
     1111111111111111111111111
     222111111"
1020 DATA"??>><<::88::?377<<5599:
     CE5:25"
1030 DATA"22222222222244222222222222
     2422"
1040 DATA"332200..,,..33++0055--.
     ?A5.25"
1050 DATA"22222222222244222222222222
     2422"
1060 DATA"MJFJMJFJROKOROKOPMH
     MPMJMOFHJKMOPROKOROKOR
     MJMRMJMOKHKOKHKOJFJOJFJ"
1070 DATA"1111111111111111111111111
     1111111111111111111111111111
     11111111"
1080 DATA"::8877<<55::?3CFCFAFAF?C
     ?C>C>C"
1090 DATA"22222222222244222222222222
     22222"
1100 DATA"..,,++00))..33?C?C>A>A<?
     <?:>:>"
1110 DATA"22222222222244222222222222
     22222"

2320 DATA"TRTRPOPOMOMOPRPRTRTR
     POPOMOMOPOPMORPTRVTWVT
     RWPOMKFJK"
2330 DATA"1111111111111111111111111
     1111111111111112112242244800"
2340 DATA"PRPRMOMOJOJOMRMR
     PRPRMOMOJOJOMOPMFJKOKH
     FHFD?"
2350 DATA"1111111111111111111111111
     1111111224422444480"
2360 DATA":::::::::::::::::3578:878:8::3"
2370 DATA"22222222222222222222222
     444480"
2400 DATAZ,Z
```

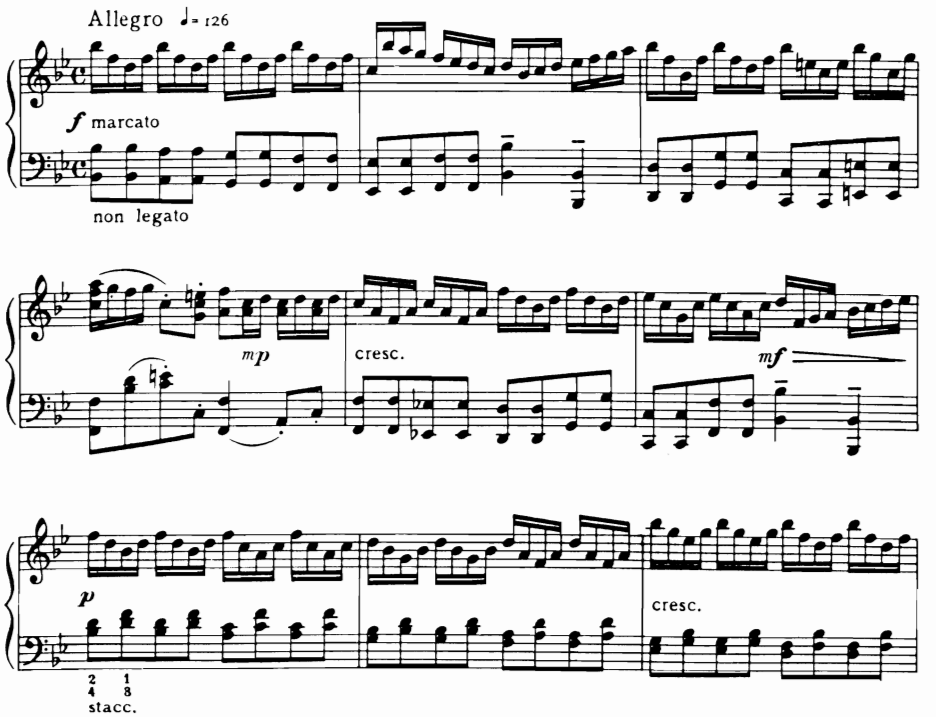

Allegro ♩= 126
f marcato
non legato
317
mp
cresc.
mf
p
cresc.
2 1
4 8
stacc.

Figure A7

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.

To use the library, load the program (from tape or diskette) 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 the routines on a separate tape or diskette file.

Note that the LIBRARY program cannot be restarted except by reloading. If interrupted in mid-stream, it may leave the 64 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: 4500-4660.

Parameters: Output: Result is delivered in X1.

Local Variables: XX$, YY$, JJ, CC$.

```
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: REM REPLACE LETTER O
     BY DIGIT 0
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
```

Driver program:

```
10 GOSUB 4500
20 PRINT "VALUE ="; X1
30 GOTO 10
```

Test results:

```
RUN
? 778
VALUE = 778
? IOI
VALUE = 101
? -34.56
VALUE = -34.56
? 45.K
NUMBERS CONSIST OF
DECIMAL DIGITS 0-9,
+,- 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: 7000-7090

Parameters: Output: Result delivered in X1.

Local Variables: PP, AA$, XX$.

```
7000 REM ROBUST NUMBER INPUT
7010 XX$ = "":PP=0
7020 GET AA$: IF AA$ = "" THEN 7020
7030 IF AA$>= "0" AND AA$ <= "9"
     THEN PRINT AA$;: XX$ = XX$ + AA$:
     PP = PP+1: GOTO 7020
7040 IF ASC(AA$) <> 20 THEN 7070:
     REM LOOK FOR DEL
7050 IF PP=0 THEN 7020: REM CAN'T
     ERASE NOTHING
7060 PRINT " [SHIFT] and [CRSR ⇐⇒] space
     [SHIFT] and [CRSR ⇐⇒] ";: PP=PP-1:
     XX$=LEFT$(XX$,PP): GOTO 7020
7070 IF ASC (AA$) <> 13 THEN 7020: REM
     LOOK FOR RETURN
7080 IF PP=0 THEN 7020:REM MUST BE
     SOME DIGITS
7090 X1=VAL (XX$): RETURN
```

Driver program:

```
10 GOSUB 7000
20 PRINT X1
30 GOTO 10
```

3. BIG LETTERS 64

Purpose: To Display the 64 characters four times their usual size.

Lines: 8000 to 8200.

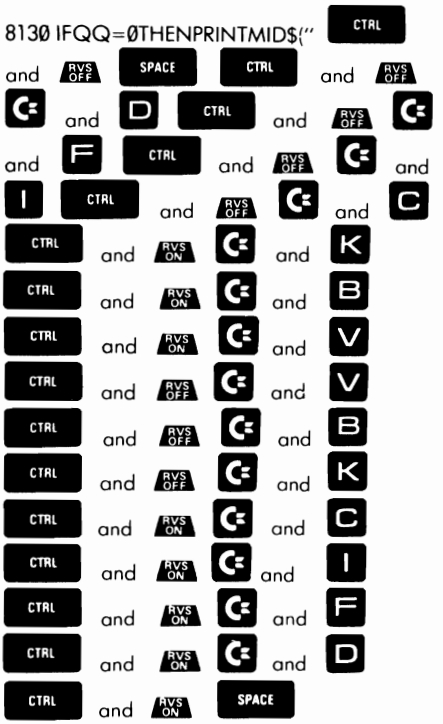
Parameters: Input: The next character to be displayed is supplied in A1$. It may be any printable character or space, RETURN, CLR/HOME, a colour code, CTRL and RVS ON or CTRL and RVS OFF.

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

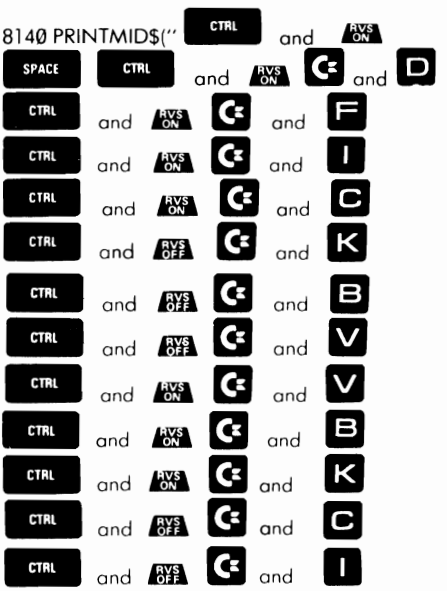
Note: QQ keeps track of the current RVS status and must not be used outside the subroutine.

```
8000 REM DISPLAY CHARACTER IN A1$
     FOUR TIMES USUAL SIZE
8010 BB = ASC (A1$)
8020 IF BB = 13 OR BB = 141 THEN PRINT
     " [CRSR↓][CRSR↓][CRSR↓]": RETURN
8030 IF BB = 18 THEN QQ = 1: RETURN
8040 IF BB = 146 THEN QQ = 0: RETURN
8050 IFBB<32THENPRINTMID$(" [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
8060 IFBB>=144 AND BB<160THENPRINT
     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
8070 AA= (BB AND 31)+0.5 * (BB AND
     128): IF (BB AND 64)=0 THEN
     AA=AA+32
8080 FOR JJ=0 TO 6 STEP 2
8085 POKE 56334,PEEK(56334) AND 254: POKE
     1,PEEK(1) AND 251
8090 KK=PEEK (53248+8*AA+JJ):
     LL=PEEK(53249+8*AA+JJ)
8095 POKE 1,PEEK(1) OR 4: POKE 56334, PEEK
     (56334) OR 1
8100 NN=64: FOR MM=0 TO 3
```

```
8110 PP = 1+8★ INT (KK/NN)+2★INT
     (LL/NN)
8120 KK=KK−INT (KK/NN)★NN:LL=LL−
     INT (LL/NN)★NN
```

8130 IFQQ=ØTHENPRINTMID$("


",PP,2);:GOTO8150

8140 PRINTMID$("


",PP,2);
8150 NN=INT(NN/4):NEXT MM
8160 PRINT"   4 times";
8170 NEXTJJ
8180 PRINT"   4 times 
4 times ";
8190 IFPEEK(211)>36THENPRINT" 
2 times"
8200 RETURN

Driver program

```
10 GET A1$: IF A1$= "" THEN 10
20 GOSUB 8000
30 GOTO 10
```

Sample run: Not reproduceable. (Try it and see.)

4. FORMATTED NUMBER

Purpose: To display a number in a controlled format.

Lines: 5000 to 5130.

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 = 0 the number is rounded to the nearest integer.

```
5000 REM DISPLAY X1 TO Y1 DECIMAL
     PLACES
5010 XX=X1: IF Y1 >0 AND ABS(XX) <=
     999999999 THEN 5050
5020 XX=XX+0.5
5030 PRINT INT (XX);
5040 RETURN
5050 IF XX <0 THEN XX=XX−0.5★10↑−
     Y1:GOTO 5070
5060 XX=XX+0.5★10↑−Y1
5070 NN$= STR$ (XX)
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 FOR JJ=1 TO Y1: PRINT "0";: NEXT JJ
5130 RETURN
```

Driver Program:

```
10 FOR J= 667 TO 670
20 FOR Y1=0 TO 3
30 X1= SQR(J)
40 GOSUB 5000
50 NEXT Y1
60 PRINT
70 NEXT J
80 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: 5700-5800.

Parameters: Input: String to be displayed in X1$.

Local Variables: XX$, PP, QQ, RR.

```
5700 REM DISPLAY X1$ WITHOUT
     SPLITTING WORDS
5710 XX$=X1$
5720 PP=LEN(XX$)
5730 IF PP<=40 THEN RR=PP:GOSUB
     5780: RETURN
5740 FOR QQ=41 TO 1 STEP -1
5750 IF MID$(XX$,QQ,1) = " " THEN
     RR=QQ-1:GOSUB 5780: XX$=
     RIGHT$(XX$, PP-QQ):GOTO
     5720
5760 NEXT QQ
5770 RR=40: GOSUB 5780: XX$=RIGHT$
     (XX$, PP-40): GOTO 5720
5780 REM INTERNAL SUBROUTINE
5790 PRINT LEFT$(XX$, RR);: IF RR<40
     THEN PRINT
5800 RETURN
```

Driver program:

```
10 X1$="THIS IS THE FIRST PART
   OF A LONG STRING TO TRY OUT "
20 X1$=X1$+"THE STRING PRINTING
   ROUTINE. IT SHOULD ARRANGE "
30 X1$=X1$+"THE WORDS PROPERLY"
40 GOSUB 5700
50 STOP
```

Sample output: Try it and see!

6. BINARY CONVERTER

Purpose: To display the binary pattern of a number in the range 0-255.

Lines: 1000-1060.

Parameters: Input: Number to be displayed in X1.

Local Variables: YY, KK, XX.

```
1000 REM CONVERT X1 TO BINARY AND
     DISPLAY
1010 YY=256: XX=X1: FOR KK=1 TO 8
1020 YY=YY/2
1030 IF XX>=YY THEN XX=XX-YY:
     PRINT "★";:GOTO 1050
1040 PRINT " ";
1050 NEXT KK
1060 PRINT: RETURN
```

Driver program:

```
10 FOR J=0 TO 9
20 X1=J: GOSUB 1000
30 NEXT J
40 STOP
```

Sample output:

7. EXTRACT SURNAME

Purpose: To extract a surname from a person's full name.

Lines: 4100-4200.

Parameters: Input: A person's name, in N1$, in any of the following forms:
J. X. SMITH
GEORGE ELLIOTT
ALVA T EDISON
WELLINGTON-COO
K. O'SHAUGNESSY

Output: The person's surname, in Y1$. The surname is defined as the unbroken sequence of letters, hyphens and apostrophes nearest the end of the string in N1$.

Examples are: SMITH
ELLIOTT
EDISON
WELLINGTON-COO
O'SHAUGNESSY

If a surname can't be found, Y1$ 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 N1$
     AND DELIVER IN Y1$
4110 JJ=LEN (N1$)
4120 IF JJ=0 THEN Y1$= "": RETURN
4130 IF MID$(N1$, JJ, 1) <"A" OR MID$
     (N1$, JJ, 1) > "Z" THEN JJ=JJ-1:
     GOTO 4120
4140 FOR KK=JJ TO 1 STEP -1
4150 CC$=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
```

Driver program:

```
10 INPUT "NAME PLEASE"; N1$
20 GOSUB 4100
30 PRINT "SURNAME IS "; Y1$
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 A1$
Index of first entry in L1; index of last entry in H1
String to be looked for in X1$.

Output: If the entry is found, M1 holds its index; if not found M1 = −1.

Local Variables: HH, LL.

```
6000 REM SEARCH ORDERED LIST IN A1$
6005 HH=H1: LL=L1
6010 IF HH<LL THEN M1 = -1: RETURN
6020 M1=INT (0·5*(HH+LL))
6030 IF X1$= A1$(M1) THEN RETURN
6040 IF X1$ < A1$(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 A1$(14)
40 FOR J=1 TO 14: READ A1$(J): NEXT J
50 INPUT "TYPE A NAME";X1$
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 A1$ (1) to A1$(N1).
Number of items in N1.

Output: Sorted list in A1$ (1) to A1$ (N1).

Local Variables: KK, DD$, SS$.

```
6500 REM BUBBLE SORT OF STRINGS IN
     A1$
6510 SS$= "NO"
6520 FOR KK=1 TO N1-1
```

```
6530 IF A1$(KK)>A1$(KK+1) THEN DD$=
     A1$(KK): A1$(KK)=A1$(KK+1):
     A1$(KK+1)=DD$: 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 A1$ (N1)
40 FOR J=1 TO N1: INPUT A1$(J): NEXT J
50 GOSUB 6500
60 FOR J=1 TO N1: PRINT A1$(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: A1$ 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: AA=XX+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
```

```
5510 REM RESULT IN C1/D1. LOCALS ARE
     JJ, KK
5520 REM ERROR IF A1 OR B1 NOT
     WHOLE NUMBERS OR IF B1 < 1
5530 IF A1 = INT(A1) AND B1 = INT(B1)
     AND B1 > 1 THEN 5550
5540 PRINT "WRONG PARAMETERS TO
     FRACTION SIMPLIFIER"; A1;B1: STOP
5550 IF A1=0 THEN C1=0: D1=1: RETURN
5560 JJ=A1: KK=B1
5570 IF A1 < 0 THEN JJ = -A1
5580 IF KK=0 THEN 5620
5590 IF JJ=0 THEN JJ=KK: GOTO 5620
5600 IF JJ>KK THEN JJ=JJ-INT(JJ/KK)*
     KK: GOTO 5580
5610 KK=KK-INT (KK/JJ)*JJ: GOTO
     5580
5620 C1=A1/JJ: D1=B1/JJ
5630 RETURN
```

Driver program:

```
10 INPUT "GIVE FRACTION"; A1,B1
20 GOSUB 5500
30 PRINT C1;"/"; D1
40 GOTO 10
```

Sample run:

```
RUN
GIVE FRACTION? 2, 4
1/2
GIVE FRACTION? 123, 456
41/152
GIVE FRACTION? 375, 1000
3/8
GIVE FRACTION? 0, 1234
0/1
GIVE FRACTION? 0·5, 0·75
WRONG PARAMETERS TO
FRACTION SIMPLIFIER
.5      .75
RUN
GIVE FRACTION? -50, 100
-1/2
GIVE FRACTION? 77, 0
WRONG PARAMETERS TO
FRACTION SIMPLIFIER
77      0
```

12. SIMULTANEOUS EQUATIONS

Purpose: To solve simultaneous equations, N1 eg. equations in N1 unknowns.

Lines: 9000-9270.

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:

```
10 DATA 3, 2, 1, 19
20 DATA 2, 7, 2, 55
30 DATA 4, 1, 4, 19
40 DIM A1(3,3), B1(3), X1(3)
50 N1=3
60 FOR J=1 TO N1: FOR K=1 TO N1:
   READ A1(J,K): NEXT K: READ B1(J):
   NEXT J
70 GOSUB 9000: REM CALL
   SUBROUTINE
80 FOR J=1 TO N1: PRINT X1(J):
   NEXT J
90 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
10	10
15	30
20	65
25	121

```
9000 REM SOLVE N1 SIMULTANEOUS
     EQUATIONS A1.X1=B1
9010 IF N1=1 THEN X1(1)=B1(1)/A1(1,1):
     RETURN
9020 FOR JJ=1 TO N1-1: REM FIND
     PIVOT
9030 DD=ABS(A1(JJ,JJ)): LL=JJ
9040 FOR KK=JJ TO N1
9050 IF ABS(A1(KK,JJ))>DD THEN DD=
     ABS(A1(KK,JJ)):LL=KK
9060 NEXT KK
9070 IF LL=JJ THEN 9120
9080 FOR KK=JJ TO N1: REM
     INTERCHANGE EQUATIONS
9090 DD= A1(JJ,KK): A1(JJ,KK)=A1(LL,
     KK): A1(LL,KK)=DD
9100 NEXT KK
9110 DD=B1(JJ): B1(JJ)=B1(LL): B1(LL)=
     DD
9120 FOR KK=JJ+1 TO N1: DD=A1(KK,
     JJ)/A1(JJ,JJ)
9130 FOR LL=JJ TO N1 : REM ELIMINATE
9140 A1(KK,LL)=A1(KK,LL)-DD*A1(JJ,
     LL)
9150 NEXT LL
9160 B1(KK)=B1(KK)-DD*B1(JJ)
9170 NEXT KK
9180 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

```
  10 INPUT "WAGES IN CENTS";W
  20 FOR J=1 TO 8
  30 READ V,N$
  40 T=INT(W/V)
  50 PRINT T;N$
  60 W=W-V*T
  70 NEXTJ
  80 STOP
1000 DATA 5000,FIFTY-DOLLAR BILL(S)
1010 DATA 1000,TEN-DOLLAR BILL(S)
1020 DATA 500,FIVE-DOLLAR BILL
1030 DATA 100,DOLLAR(S)
1040 DATA 25,QUARTER(S)
1050 DATA 10,DIME(S)
1060 DATA 5,NICKEL
1070 DATA 1,CENT(S)
```

Experiment 16.2A

```
  10 FOR K=1 TO 12
  20 READ M$
  30 PRINT M$
  40 NEXT K
  50 STOP
1000 DATA JANUARY,FEBRUARY,MARCH,
     APRIL
1010 DATA MAY,JUNE,JULY,AUGUST
1020 DATA SEPTEMBER,OCTOBER,NOVEMBER,
     DECEMBER
```

Experiment 16.2B

```
  10 INPUT D,M,Y
  20 FOR J=1 TO M
  30 READ M$
  40 NEXT J
  50 PRINT D;M$;Y
  60 RESTORE
  70 GOTO 10
1000 DATA JANUARY,FEBRUARY,MARCH,
     APRIL
1010 DATA MAY,JUNE,JULY,AUGUST
1020 DATA SEPTEMBER,OCTOBER,NOVEMBER,
     DECEMBER
```


Experiment 16.3

```
  10 T=0
  20 S=0
  30 PRINT" SHIFT and CLR HOME "
  40 READA$
  50 IF A$="END" THEN 240
  60 READ B$
  70 T=T+1
  80 J=1
  90 PRINTA$
 100 PRINT
 110 INPUT X$
 120 IF X$=B$ THEN 200
 130 IF J=3 THEN 170
 140 J=J+1
 150 PRINT"WRONG. TRY AGAIN"
 160 GOTO 90
 170 PRINT "THE ANSWER IS"
 180 PRINT B$
 190 GOTO 40
 200 PRINT "THAT'S RIGHT!"
 210 IF J>1 THEN 40
 220 S=S+1
 230 GOTO 40
 240 PRINT "YOU GOT";S;"RIGHT"
 250 PRINT "FIRST TIME"
 260 PRINT"OUT OF";T;"QUESTIONS"
 270 STOP
 280 DATA WHO COMPOSED THE MESSIAH,
     HANDEL
 290 DATA HOW MANY SYMPHONIES DID
     BEETHOVEN WRITE,NINE
 300 DATA WHO WROTE THE OPERA CARMEN,
     BIZET
 310 DATA WHAT DID PAGANINI PLAY,VIOLIN
 320 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

```
N$<>"JONES" AND N$<>"SMITH" AND
N$<>"BROWN"
X>15 OR X<4
```

Experiment 17.2B

```
 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 17.2C

```
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 1000:REM MAKE SOUND
150 NEXT R
160 STOP
1000 REM SUBROUTINE TO MAKE PIP SOUND
1010 VV = 212*256
1020 POKE VV + 24, 15: POKE VV,0
1030 POKE VV+1,80: POKE VV+5,9
1040 POKE VV+4,0: POKE VV+4,33
1050 FOR MM = 1 TO 800: NEXT MM
1060 RETURN
2000 REM SUBROUTINE TO CHANGE BORDER
     COLOUR
2010 IF Z=X+Y THEN POKE 53280,4: S=S+1:
     GOTO 2030:REM RIGHT ANSWER —
```

```
     BORDER PURPLE
2020 POKE 53280,0: REM WRONG ANSWER
     — BORDER BLACK
2030 FOR KK=1 TO 800:NEXT KK
2040 POKE 53280,14: REM RESTORE INITIAL
     COLOUR
2050 RETURN
```

Experiment 18.1B

```
 10 PRINT" [SHIFT] and [CLR HOME] COUNTING
    TEST"
 20 PRINT"COUNT THE PIPS"
 30 S=0
 40 FOR A=1 TO 10
 50 FOR T=1 TO 5000:NEXT T:REM WAIT A BIT
 60 X=INT(9*RND(0)+1)
 70 FOR J=1 TO X
 80 GOSUB 1000
 90 NEXT J
100 INPUT Z
110 IF Z=X THEN GOSUB 5000:GOSUB 2000:
    GOSUB 4000:GOTO130
120 GOSUB3000:GOSUB 4000
130 NEXT A
140 PRINT"THAT'S";S;"RIGHT"
150 STOP
1000 REM SUBROUTINE TO MAKE PIP SOUND
1010 VV = 212*256
1020 POKE VV + 24, 15: POKE VV,0
1030 POKE VV+1,80: POKE VV+5,9
1040 POKE VV+4,0: POKE VV+4,33
1050 FOR MM = 1 TO 800: NEXT MM
1060 RETURN
2000 REM SUBROUTINE TO CHANGE BORDER
     COLOUR PURPLE—RIGHT ANSWER
2010 IFZ=XTHENPOKE53280,4:S=S+1:
     RETURN
3000 REM SUBROUTINE TO CHANGE BORDER
     COLOUR BLACK—WRONG ANSWER
3010 POKE 53280,0:RETURN
4000 REM SUBROUTINE TO RESTORE INITIAL
     COLOUR
4010 FORKK=1TO800:NEXTKK
4020 POKE 53280,14:RETURN
5000 REM MYSTERY
5010 VV=212*256: POKE VV+24,15
5020 POKE VV,0: POKE VV+5,45
5030 POKE VV+4,0: POKE VV+4,33
5040 FOR MM=50 TO 120 STEP 0.5
5050 POKE VV+1,MM
5060 NEXT MM
5070 POKE VV+24,0: RETURN
```

Experiment 18.2

```
10 PRINT " [SHIFT] and [CLR HOME] ";
20 FOR X1=0TO35
30 C1$=" [CTRL] and [RED] "
40 GOSUB 500
50 FOR T=1TO150 :NEXT T
60 C1$=" [CTRL] and [BLU] "
70 GOSUB500
80 NEXT X1
90 GOTO 90
500 REM MONSTER
510 PRINT" [CLR HOME] [CRSR↓] [CRSR↓] ";C1$;
520 IF X1=0THEN540
530 FORJJ=1 TO X1:PRINT" [CRSR→] ";:NEXT JJ
540 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] ",
550 RETURN
```

Experiment 18.3

```
 10 PRINT" [SHIFT] and [CLR HOME] "
 20 X1=1:Y1=6:N1=15: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

```
  10 INPUT"FIRST FRACTION";P,Q
  20 INPUT"SECOND FRACTION";S,T
  30 A1=P*T+Q*S
  40 B1=Q*T
  50 GOSUB 5500
  60 PRINT "RESULT =";C1;"/";D1
  70 STOP
5500 REM REDUCE FRACTION R1/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
```

Experiment 19.2B

```
  10 INPUT"FIRST FRACTION";P,Q
  20 INPUT"SECOND FRACTION";S,T
  30 A1=P*T+Q*S
  40 B1=Q*T
  50 GOSUB 5500
  60 PRINT "RESULT =";C1;"/";D1
  70 STOP
5500 REM REDUCE FRACTION A1/B1 TO
     LOWEST TERMS, USING DIVISION NOT
     SUBTRACTION
5510 REM RESULT IN C1/D1 LOCALS ARE
     JJ,KK,LL
5520 REM ERROR IF A1 OR B1 NOT WHOLE
     NUMBERS OR IF B1<1
5530 IF A1=INT(A1) AND B1=INT(B1)AND
     B1>=1 THEN 5550
5540 PRINT"WRONG PARAMETERS—":PRINT
     A1;B1:STOP
5550 IF A1=0THEN C1=0:D1=1:RETURN
5560 LL=1:IFA1<0THEN LL=-1:A1=-A1
5570 JJ=A1:KK=B1
5580 IFKK=0THEN 5620
5590 IFJJ=0THEN JJ=KK:GOTO5620
5600 IFJJ>KK THEN JJ=JJ-INT(JJ/KK)*KK:
     GOTO5580
5610 KK=KK-INT(KK/JJ)*JJ:GOTO 5580
5620 C1=LL*A1/JJ:D1=B1/JJ
5630 RETURN
```

Experiment 19.3

```
  10 INPUT"THREE NUMBERS";A1,B1,C1
  20 GOSUB 1000
  30 PRINT "LARGEST IS";X1
  40 GOTO 10
1000 REM FIND LARGEST OF A1,B1,C1
1010 REM AND DELIVER RESULT IN X1
1020 X1=A1
1030 IF X1<B1 THEN X1=B1
1040 IF X1<C1 THEN X1=C1
1050 RETURN
```

Experiment 19.4

Solution in Appendix B, BIG LETTERS 64

UNIT: 20

Experiment 20.1

```
10 DIM W$(100)
20 N=0
30 INPUT"NAME";X$
40 IF X$="ZZZZ" THEN 60
50 N=N+1:W$(N)=X$:GOTO30
60 FOR J=N TO 1 STEP -1
70 PRINT W$(J)
80 NEXT J
90 STOP
```

Experiment 20.2A

```
 10 DIM T$(40)
 20 FOR J=0 TO 40
 30 READ T$(J)
 40 NEXT J
 50 DATA NIL,I,II,III,IV,V,VI,VII,VIII,IX,X
 60 DATAXI,XII,XIII,XIV,XV,XVI,XVII,XVIII,XIX,
    XX
 70 DATAXXI,XXII,XXIII,XXIV,XXV,XXVI,XXVII,
    XXVIII,XXIX,XXX
 80 DATAXXXI,XXXII,XXXIII,XXXIV,XXXV,XXXVI,
    XXXVII,XXXVIII,XXXIX,XL
100 PRINT"GIVE TWO NUMBERS"
110 INPUT X$,Y$
120 A1$=X$:GOSUB 1000:X=B1
130 A1$=Y$:GOSUB1000:Y=B1
140 Z=X+Y
150 IF Z>40 THEN PRINT"RESULT EXCEEDS
    CAPACITY":STOP
160 PRINT"SUM = ";T$(Z)
170 STOP
1000 REM CONVERT WORD A1$ INTO ROMAN
    NUMBER B1
1010 FOR JJ =0 TO 40
1020 IF A1$=T$(JJ) THEN 1050
1030 NEXT JJ
1040 PRINT "NO ENTRY FOUND": STOP
1050 B1=JJ
1060 RETURN
```

Experiment 20.2B1

```
  7 REM SOLUTION WITH ARRAYS
 10 DIM N$(20),T$(20)
 20 FORJ=1 TO 20
 30 READ N$(J),T$(J)
 40 NEXTJ
 50 INPUT "NAME";X$
 60 FOR J=1 TO 20
 70 IF X$=N$(J) THEN PRINT X$;"'S PHONE IS
    ";T$(J):PRINT:GOTO 50
 80 NEXT J
 90 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=0 TO 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
  40 DATABAIN,BEAVIS,BOWEY,BURNS,
      CLARK,FLEMING
  50 DIMA1$(26)
  60 FORJ=1 TO 26:READA1$(J):NEXT J
  70 N1=26:GOSUB6500
  80 FOR J=1 TO 26
  90 PRINT A1$(J)
 100 NEXT J
 110 STOP
6500 REM BUBBLE SORT N1 ITEMS IN A1$(1)TO
      A1$(N1)
6510 SS$="N"
6520 FOR KK=1 TO N1-1
6530 IFA1$(KK)>A1$(KK+1)THENDD$=A1$
      (KK):A1$(KK)=A1$(KK+1):A1$(KK+1)=
      DD$:SS$="Y"
6540 NEXT KK
6550 IF SS$="Y"THEN 6510
6560 RETURN
```

Experiment 22.3A

```
  10 DATAADAMS,27
  20 DATABRIGGS,66
  30 DATACHILVERS,29
  40 DATADALE,38
  50 DATACOLIN,12
  60 DATAMACSNOOT,67
  70 DATAWILSON.96
  80 DATATHOMSON,53
  90 DATAWILMOTT,31
 100 DATABAIN,42
 110 DATAMUNDY,64
 120 DATAKRESTIN,85
 130 DATAMCILDOWIE,10
 140 DATAWRAITH,99
 150 DATAGREEN,72
 160 DATAGREENE,52
 170 DATASHEPHERD,53
 180 DATAHUTCHISON,64
 190 DATABLACK,45
 200 DATABAXTER,1
 210 DATASMYRL,99
 220 DATAFLASHMAN,2
 230 DATAMORRIS,75
 240 DATAELLIS,42
 250 DATAMATTHEWS,66
 260 DATAFOLEY,91
 270 DATACOLLINS,36
 280 DATADANIELS,93
 290 DATAJACKSON,77
 300 DATAPEIRCE,78
 310 DATADEWEY,34
 320 DATAMACGREGOR,15
 330 DATAEASON,69
 340 DATAPARSONS,6
 350 DATATHATCHER,45
 360 DATACLAYMORE,85
 370 DATAO'FLAHERTY,66
 380 DATABUNN,5
 390 DATASULLIVAN,85
 400 DATAGILBERT,41
 410 DATAZZZZ,0
 500 N1=100:REMMAXNO.OF STUDENTS
 510 DIM S$(N1),M(N1),A1(N1)
 520 FOR J=1TON1
```

```
530 READ S$(J),M(J)
535 IF$ (J)="ZZZZ" THEN GOTO 560
540 A1(J)=M(J)
550 NEXT J
560 N1=J-1: REM NOW SORT MARKS IN A1
570 GOSUB 6000
580 REM FIND PASS-MARK (THREE QUARTERS-
    WAY DOWN LIST)
590 P=A1(INT(0.75*N1+1))
600 PRINT"PASS-MARK IS";P
610 REM NOW DISPLAY STUDENTS WHO PASS
620 FOR J=1 TO N1
630 IF M(J)>=P THEN PRINT S$(J),M(J)
640 NEXT J
650 STOP
6000 REM QUICKSORT:SORTS N1 ELEMENTS
    OF A1
6010 IFSS=1THEN6030
6020 DIMSS%(100):SS=1:REM DECLARE STACK
6030 AA=1:BB=N1:SS%(0)=1:PP=1
6040 XX=AA:YY=BB:ZZ=A1(BB)
6050 IFXX>=YYTHEN6090
6060 IFA1(XX)<=ZZTHENXX=XX+1:GOTO
    6050
6070 IFA1(YY)>=ZZTHENYY=YY-1:GOTO
    6050
6080 DD=A1(YY):A1(YY)=A1(XX):A1(XX)=
    DD:GOTO6050
6090 A1(BB)=A1(XX):A1(XX)=ZZ
6100 IFXX-AA<=1THEN6140
6110 SS%(PP)=XX:SS%(PP+1)=BB:SS%(PP+2)
    =2:PP=PP+3
6120 BB=XX-1:GOTO6040
6130 PP=PP-3:XX=SS%(PP):BB=SS%(PP+1)
6140 IFBB-XX<=1THEN6170
6150 SS%(PP)=3:PP=PP+1:AA=XX+1:GOTO
    6040
6160 PP=PP-1
6170 ONSS%(PP-1)GOTO6180,6130,6160
6180 RETURN
```

Experiment 22.3B

```
10 REM FIRST PART OF LIFE PROGRAM
20 DIMX$(9,9),Y$(9,9)
30 PRINT" [SHIFT] and [CLR HOME] TYPE
   STARTING PATTERN"
40 PRINT"IN ★'S AND SPACES,"
50 PRINT"USING CURSOR CONTROLS"
60 PRINT"TO GET IT INSIDE THE"
70 PRINT"BOX. USE RETURN"
73 PRINT "TO FINISH EACH ROW"
77 PRINT"(EVEN EMPTY ONES)."
80 PRINT" 3 spaces [C=] and [A]
   [SHIFT] and [*] 9 times [C=] and
   [S] "
90 FORJ=1 TO 9
100 PRINT" 3 spaces [SHIFT] and [-]
    9 spaces [SHIFT] and [-] "
110 NEXTJ
120 PRINT" 3 spaces [C=] and [Z]
    [SHIFT] and [*] 9 times [C=] and
    [X] "
130 PRINT" [CLR HOME] [CRSR] 8 times",
140 FOR J=1 TO 9
150 T$=" ":INPUT T$
160 FORK=2TO 10
170 X$(J,K-1)=MID$(T$,K,1)
180 NEXTK,J
190 REM DISPLAY CURRENT POSITION
200 PRINT" [SHIFT] and [CLR HOME] [CRSR] 3 times
    3 spaces [C=] and [A] [SHIFT]
    and [*] 9 times [C=] and [S] "
210 FORJ=1 TO 9
220 PRINT" 3 spaces [SHIFT] and [-] ";
230 FORK=1TO9:PRINTX$(J,K);:NEXTK
240 PRINT" [SHIFT] and [-] "
250 NEXTJ
260 PRINT" 3 spaces [C=] and [Z]
    [SHIFT] and [*] 9 times [C=] and
    [X] "
270 FOR J=2 TO 8
280 FOR K=2 TO 8
290 T=0
300 IFX$(J-1,K-1)="★"THEN T=T+1
310 IFX$(J-1,K)="★"THEN T=T+1
320 IFX$(J-1,K+1)="★"THEN T=T+1
330 IFX$(J,K-1)="★"THEN T=T+1
340 IFX$(J,K+1)="★"THEN T=T+1
350 IFX$(J+1,K-1)="★"THEN T=T+1
360 IFX$(J+1,K)="★"THEN T=T+1
370 IFX$(J+1,K+1)="★"THEN T=T+1
380 Y$(J,K)=X$(J,K)
390 IF T<2 OR T>3THEN Y$(J,K)=" "
400 IF T=3THENY$(J,K)="★"
410 NEXT K,J
420 FORJ=2 TO 8
430 FORK=2 TO 8
440 X$(J,K)=Y$(J,K)
450 NEXT K,J
460 GOTO190
```

UNIT:24

Experiment 24.1

```
 10 PRINT"  SHIFT  and  CLR HOME  ";
 20 X=20:Y=12
 30 POKE 1024+40*Y+X,160
 40 POKE 55296+40*Y+X,0
 50 GET A$:IF A$="" "THEN50
 60 IF ASC(A$)<> 133 THEN 80
 70 IF Y>0THEN Y=Y-1:GOTO30
 80 IFASC(A$)<>134 THEN 100
 90 IF X<39 THEN X=X+1:GOTO30
100 IF ASC(A$)<>135THEN120
110 IF Y<24 THEN Y=Y+1:GOTO30
120 IF ASC(A$)<>136THEN GOTO50
130 IF X>0THEN X=X-1:GOTO30
140 GOTO50
```

Experiment 24.2A

```
 10 DEF FNA(X)=X↑3+(X+7)↑2-100
 50 FOR J=2TO3STEP0.1
 60 PRINT J;FNA(J)
 70 NEXT J
 80 STOP
100 REM BEST ESTIMATE FOR SOLUTION IS
    ABOUT 2.33
```

Experiment 24.2B

```
10 DEF FNA(X)=X↑3+(X+7)↑2-100
20 DEF FNB(X)= 3*X↑2 +2*(X+7)
30 X=2
40 Y=FNA(X)
50 PRINT "X=";X
60 PRINT"Y=";Y
70 IF ABS(Y)>0.0000001 THEN X=X-Y/FNB(X):
   GOTO40
80 PRINT"SOLUTION IS";X
90 STOP
```

Experiment 24.3

Solutions are given as programs

EXPT 24.3A (T)

and

EXPT 24.3B (T)

if you are using cassette tape, or

EXPT 24.3A (D)

and

EXPT 24.3B (D)

for diskette.

Experiment 24.4

Solutions are given as programs

DATEPROG (T)

(for cassette tape), or

DATEPROG (D)

for diskette.

INDEX

INTRODUCTION TO BASIC PART 2

Tape 1

UNIT16QUIZ64
UNIT17PROG64
PICTURE64
BIGLETTERS64
UNIT20QUIZ64
UNIT21QUIZ64
QUICKSORT
LIFESTART
MONDRIAN
TONGUE
WASPS64
MAKENAMES(T)
GRAFFS
DUNGEON
SDP
MONSPR
LINEAR
CIRCULAR1
CIRCULAR2
BOUNCE
BUS

Tape 2

COSPRED
SOLAR
GROTTY
SPRMAC
PLANETS
DUBLIN
TUNE PLAYER
SHEBA
LIBRARY
EXPT24.3A(T)
EXPT24.3B(T)
DATEPROG(T)

NOTE: All the programs are recorded again on the reverse of the tape.

commodore COMPUTER

Commodore Business Machines, Inc.
1200 Wilson Drive • West Chester, PA 19380

Commodore Business Machines, Limited
3370 Pharmacy Avenue • Agincourt, Ontario, M1W 2K4