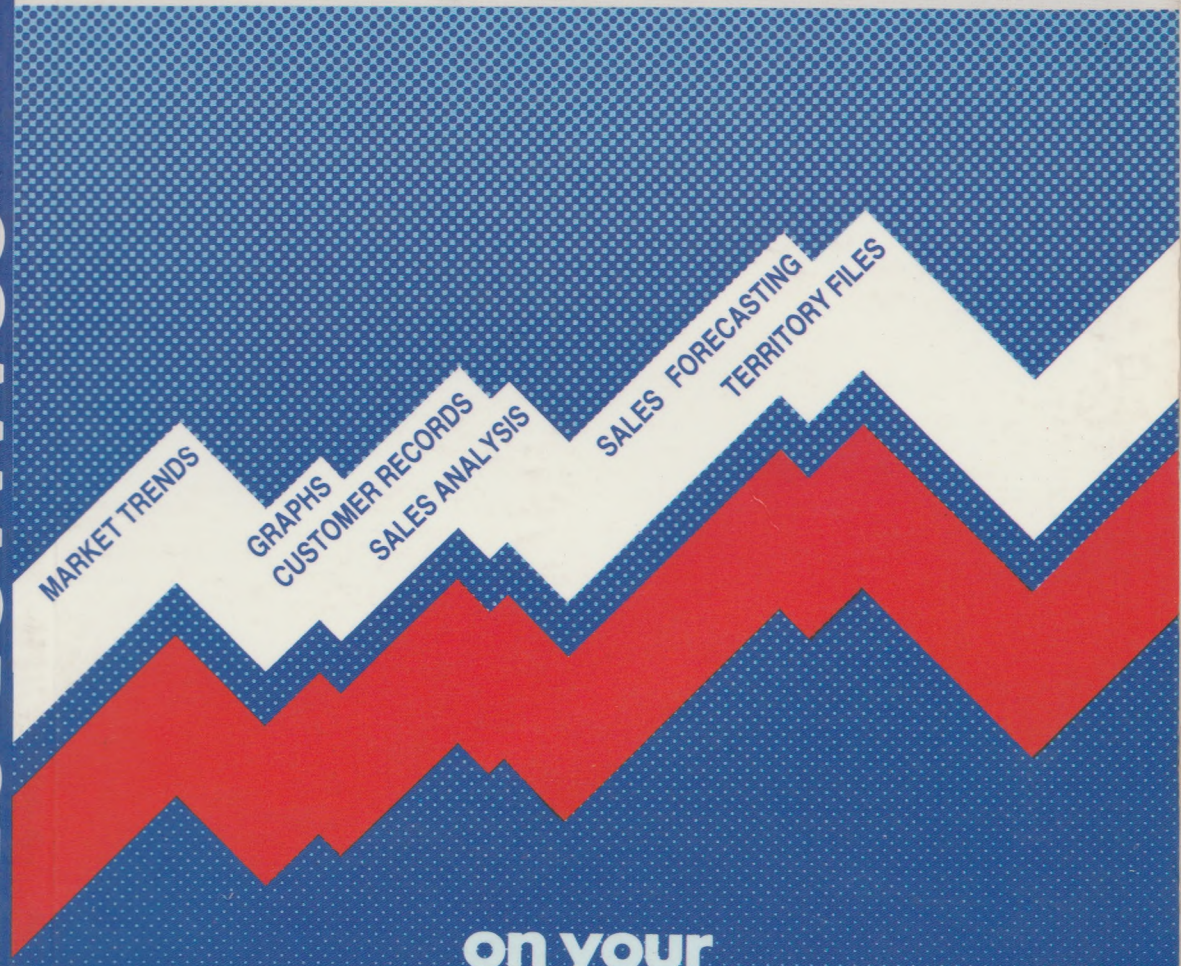
BUSINESS PROGRAMMING



on your

COMMODORE

PETER JACKSON
WITH
PETER GOODE

BUSINESS PROGRAMMING on your COMMODORE

PETER JACKSON
WITH
PETER GOODE

PHOENIX

BUSINESS PROGRAMMING
on your
COMMODORE 64

BUSINESS PROGRAMMING
on your
COMMODORE 64

PETER JACKSON
with
Peter Goode

Phoenix Publishing Associates
Bushey, Herts

First published in Great Britain in 1984 by
PHOENIX PUBLISHING ASSOCIATES LTD
14, Vernon Road, Bushey, Herts. WD2 2JL

ISBN 0 9465 7619 X

Printed in Great Britain by
The Garden City Press, Letchworth
Cover design and graphics by
Denis Gibney Graphics
Chesham and Chorleywood
Typesetting by
Prestige Press (UK) Ltd
Chesham, Bucks

CONTENTS

INTRODUCTION

This book is for people interested in ways of applying personal computers to real life business problems. It is set in the context of sales and marketing management where, despite the relative lack of micro-computer literature, there is plenty of scope for finding better solutions to well known problems. Since some sort of selling activity takes place in all businesses, the book will also be of interest to general managers and to those who work alongside sales and marketing personnel.

The early chapters which comprise an introduction to Basic programming and a discussion of some of the principles of programming, lead in to the main part of the book. This is made up of five chapters, each of which presents a program related to a particular aspect of sales or marketing management. The chapters follow the same format, the problem is introduced, the program is described in general terms and then in detail with references to the complete listing at the end of the chapter.

Chapters are ordered so that, where appropriate, sections from the earlier programs are incorporated in the rather longer ones which follow. This arrangement avoids unnecessary duplication of text but means that the book should be read in chapter order on the first occasion.

The last two programs in the book are databases and require a disk drive.

The final chapter contains an outline description of a comprehensive management package. Each sub-section of the package has already been the subject of an earlier chapter. The chapter discusses and outlines how these sub-sections can be put together into the complete package. The reader is invited to carry on where the book leaves off.

1

COMPUTERS
Friends or Foes?

How do you feel about computers? If you are a salesman, you will have mixed feelings. Your main contact with a computer is likely to be something called "Actual sales versus target. Territory 12. Representative Joe Soap" which you meet at the monthly performance review. The review takes place between you, the manager and a 3 inch high pile of green and white paper. The manager turns the paper round and round trying to find out where it starts, unable to cope with pages which are all stuck together and half of which are upside down. Eventually he starts "The computer says that you are under target on ---------".

Maybe the factory is run with a computer. If so, the customer you know as The Bridgend Gasket Company will be listed as **BRDGD GSKT** and have a number like **E67043263.** When you want to know what has happened to their last order, no one will talk to you unless you remember the customer's computer name. Once over this problem, they look him up in another great pile of paper (or, if they are on the leading edge of technology on a television screen). They tell you that the order is due to be dispatched at the end of week 621. Once, long ago, you passed this information straight on to the customer and you can still remember the harshness with which he pointed out that there are usually only 52 weeks in the year. Now you know that 621 is in fact the 21st June because that is the way the Americans write the date!

Then of course there is the stop list. What fun you had telling the buyer at Leyfords that his line was shut down because they had not paid for the last three months supplies. "I'm sure the accounts department could not have made a mistake but -------- perhaps I should check -------. It could be a problem with the computer --------
"

Yes, most salesmen have mixed feelings about computers.

But there is another side. You walk through the drawing office and find that most of the high stools and drawing boards have gone. The draftsmen have got computer terminals and drawings are appearing from automatic drafting machines. Down in the machine shop, they are sticking tape cassettes into the machine tools and then leaving them to work themselves. The accountants no longer add anything up – it is all done on the computer. Maybe half of you thinks you are missing out on something good as you write the customer's name and address at the top of your call report for the hundredth time, or sort through the record cards you keep in a shoebox.

If you are a Sales Director, the problem is more serious because your colleagues are the people in charge of all this new equipment. The Finance Director has figures on everything and they are right because they all come out of a computer. The Production Director also has a computer. His computer is not quite as right as the Finance Director's when it comes to money but his physical quantities are 100% correct because they come straight from production control. Even the Personnel Director has a thing he calls a package which gives information on productivity and something called added value.

At the monthly board meeting you normally discuss why the month's results differ from the forecast ("made by the whole Management Team" stresses the M.D.). Surprise, surprise it turns out that orders are down on forecast and worse still, the mix was such that production were unable to make even the reduced sales requirement. Prices seem to have drifted lower and the Finance Director is convinced it is because something is amiss with the quantity discounts. You counter by saying that you have not yet seen all the sales reports for last month but orders are always poor in August and indeed the budget is supposed to take this into account. You ask what has happened to last month's backlog and even mention a few customers who you know are screaming about the lateness of their orders. All to no avail, your "for instances" and "I remembers" are no match for the piles of solid data and the incontrovertible facts at your colleagues' fingertips. Once again the meeting closes with a pretty clear signal from the M.D. that he is none too pleased with the sales department.

SALESMEN AND COMPUTERS

There are a number of reasons why the Data Processing revolution seems to have by-passed selling.

– Until recently computers were big, expensive things which lived in offices. You had to be in the office to use them. Good salesmen stay away from the office.

– The sort of data most useful to sales people is fundamentally different from that used by accountants, who have traditionally had control over Data Processing. Sales are about trends over time. Salesmen see products in terms of the markets they serve, not as things which go through the same production processes. They look for buying patterns and, for example, want to separate customers who buy once a month from those who buy once a year. Sales people need data which goes back a long way in time so they can look for signs that market share is changing, that new products are making inroads and customers have stopped buying. Accountants, on the other hand are most interested in last month. They will report the previous month and cumulative year-to-date for comparison but fundamentally once a month is over, the figures are relegated to history, at least until the year end.

– Computers are normally installed to save cost by eliminating repetitive clerical labour. If better information is also obtained then that is a welcome spin off but it is rare for it to be the main justification for the investment. Outside the order entry department, a sales office does not offer much scope for reducing costs by using computers.

– Sales personnel tend not to be numerate. In their career development, they rarely get sent on the sort of training courses where their colleagues are introduced to computers and their application.

But all this is beginning to change. As everyone knows, computers are getting less expensive and, even more important for salesmen, they are now portable. Managements are more prepared to buy computers to improve performance rather than reduce cost. The idea of equipping key personnel with their own computer is becoming commonplace in the technical field. There

certainly are problems with software but they are not insuperable and there is little doubt that software will be written once the demand is established.

But still salesmen are very much behind in the trend. The main problem, lies in the "computer illiteracy" of the "average" sales person. Engineers are provided with their own machines mainly because they look as if they will make good use of them. Now, if the truth is told, few non-specialist engineers are computer experts until they get their hands on their own machine. The important thing is that they know enough to get started – by and large, salesmen don't.

COMPUTER AMATEURS VERSUS PROFESSIONALS

Ever since computers were invented they have been shrouded in mystery for all but a privileged few. No one outside the circle of experts has been able to say anything about computers without being overwhelmed by a torrent of incomprehensible jargon. Fierce executives well able to take on the cleverest of their subordinates tend not to be around when a row is brewing in "D.P.". If, as can happen, a new computer system takes twice as long to enter an order as it used to take to do it by hand, this is accepted as the price to be paid for progress and the reason for spending even more money. Incompetence may be suspected but it is a brave man who sets out to prove it. And why? The reason is obvious. Unlike virtually all other business activities, the average person has not the slightest idea what computers are supposed to do. And that is saying something!! After all most people don't know the first thing about selling but it does not stop them passing opinions about the efficiency of the sales force.

Perhaps the most valuable benefit to come from the widespread use of personal computers is that ordinary folk will be able to pass judgment on the work of computer professionals. This process is starting: you occasionally hear unfavourable comparisons being made between the screen display produced by "professional" software as compared with computer games. People are beginning to ask why the customer name has to be compressed into 12 characters, why the column headings cannot be put at the top of each page, and so on . . . . . .

Greater familiarity will also encourage managers to get involved in systems analysis – that is when all the main characteristics of the system are sorted out and specified before the programs are written. The traditional approach is quaintly democratic. The computer expert goes around asking people what they would like but since nobody has the slightest idea what is available, the answers are meaningless. For the expert, the great advantage of the approach is that he can be sure that if he asks 20 people what they want, he will get 20 different answers. He is therefore free to install what HE thinks they need. Furthermore if the system fails to work he can go through his notes and find some obscure requirement which he can claim as the root cause of all the difficulty. No one installs anything other than a computer in this amateurish way and the nonsense will stop just as soon as line managers take proper command of what is going on.

PERSONAL COMPUTING AND THE SALESMAN

Quite apart from the benefits to be gained from being able to make use of and to influence the computing activities of others, salesmen are particularly well placed to use what is called "personal" computing. The distinction between "personal" computing and "data processing" is not really to do with machines – it is about what is done with the machine. A small firm which uses a micro-computer to calculate its payroll is "processing data". Personal computing is about using computers to improve the way an individual does his job – particularly where the job involves decision taking and judgment.

Salesmen can make their job a lot easier if they use information properly. Sales training emphasises the importance of knowing the market, the customer's business, the competition etc. Equally the main aims of sales management are to plan calling programs so as to get an optimal balance between coverage and cost, to identify and follow up key prospects, to analyse order patterns, to detect changes in buying pattern etc.

There are many manual record systems designed to help in the processes of keeping informed but they are all time consuming and repetitive. On the other hand, this is just the sort of thing that can be done on a small computer.

With the recent appearance of really small portable computers, it is now possible to give salesmen the ability to work out a complicated quotation or a series of alternatives "right there in the customer's office". Obviously there are pitfalls but the benefits are enormous if they can be overcome.

At a more mundane level, it is now possible to write a good quality letter at home using a computer as a word processor. Reports can be done in the same way and if required they can be stored on "floppy disc" and posted into the office for printing. For the brave, there are devices which will send the data down a telephone line straight into another computer at the office.

2

Basic Programming

Introduction:

Before you can start programming, you must become familiar with two different sets of instructions:

– The programming language

and

– The operating system

Programming languages (of which Basic is one) are systems of instruction which enable a computer to do complex tasks. The program is the link between the rather limited abilities of the computer and the complexities of the real world. Indeed it is the ability to be programmed which distinguishes a computer from its close relative, the electronic calculator.

The second set of instructions, the operating system, has the entirely different purpose of controlling the interface between the computer and its peripherals. That is, it is concerned with starting and stopping the machine, loading and saving data, routing the output to the monitor, printer etc. There must be a link between the operating system and the programming language so that operating system instructions can be given directly by the program but apart from this the two systems are very different. Normally, the operating system is tailored for a particular machine whilst the programming language is essentially the same, regardless of the machine it goes on.

At the early stages of getting to know your computer, you may well find that the operating system is more of a problem than the programming language. In part this is because the programming language has been around a lot longer than the operating system and an enormous amount of development work has been put into it whereas the operating system will have been written within the last few years. A more fundamental reason however is that the operating system deals with the interface between the "pure" electronics of the computer and the hybrid electro-mechanical world of keyboards, printers and tape recorders. In the case of personal computers, the problems are exacerbated by the fact that many of the peripheral devices were not originally designed to be part of a computer at all. There is no easy answer to these problems – it is not a matter of clever mathematics or logical thinking; it is much more akin to finding out how to work a new washing machine – a task which is easy or difficult depending on who designed it and who wrote the operating manual.

Programming languages are entirely different. In the case of Basic, there are millions of people around the world who use the language on both large and small machines. The language is supported by an extensive literature and considerable effort has been put into devising quick and effective ways of teaching people to use it. In addition to the instructions sold with the computer, there are many excellent manuals which give comprehensive descriptions of the commands together with examples of how they should be used and what happens if they are used incorrectly.

This chapter is not intended to repeat the contents of a full Basic programming manual, but rather to highlight some parts of the language which play a particularly important role in the programs described in later chapters.

Fundamental ideas:

The comparison between computers and calculators which has already been mentioned, provides a convenient start on ground familiar to everyone. In their ability to do arithmetic, there is little difference between the two devices. Both of them will add, subtract, multiply, divide, cope with decimals and negatives etc.

Some calculators have memories which allow you to enter a number and then recall it for use in a subsequent calculation. Computers also have memories but their memories can store not one but hundreds of different numbers. Imagine a calculator with many different memories – you put in a number, store it, put in another number, do a calculation, store the answer in a different memory, retrieve the first number, do another calculation . . . . . . How do you remember which number is in which store? The answer is with considerable difficulty unless you invent some system for organising and controlling the memory and its contents. Computers have such a facility, calculators do not.

Now it is the computer's large memory capacity which allows it to store not only data which has been fed in but also a list of instructions, the program, saying what is to be done with the data. Furthermore, this memory facility enables both data and program instructions to be transferred to and from the computer and such peripherals as tape recorders and printers.

To summarise, it is the existence of a sizeable memory which distinguishes a computer from a calculator and the whole business of programming and giving operating system instructions is concerned with the way in which this memory is controlled and organised.

Variables

Memory is made up of hundreds of individual storage elements (think of them as pigeon holes) in which data can be stored. The storage system is controlled by giving each pigeon hole a name. This name defines the pigeon hole and, once given, the name cannot be changed (unless you set everything back to zero and start all over again). A variable is named and assigned its initial value by means of the **LET** command:

```
LET K = 20
```

This command names (or "labels") a vacant pigeon hole as **K** and assigns it a value of 20. If you refer to **K** in subsequent discussions with the computer, it goes off and finds the pigeon hole labelled **K**.

So for example, if you tell it to:

PRINT K

You get the response:

20

The computer is equally happy to assign a letter to a pigeon hole but it must have some way of distinguishing a letter which is to go into memory from another letter which is a pigeon hole label. The distinction is made by a convention which requires that letters destined for a memory location must be placed between quotation marks like this: **"K"**. Further, the name of the variable must be followed by a **$** sign to distinguish it from variables which have a numeric value. Letter variables are known as **STRINGS** and they are defined in the following way:

LET K$ = "A"

Now if you enter the instruction:

PRINT K$

You get the response:

A

So proving that **K** is not the same as **K$**.

Arithmetic and algebra:

Computers and calculators handle mathematics in much the same way. If you wanted to use a calculator to multiply 100 x 50 and then divide the answer by 25, you would first enter 100 then an X sign, then 50, then ÷ sign, then 25 and the answer would appear.

With a computer, all the instructions are entered before calculation takes place. This means that the complete set of instructions must be entered into the computer before the calculation can start and once the computer does start, the instructions cannot be altered.

Complete instructions from beginning to end are needed. The variables must be defined, assigned opening values and a list provided of all the operations which are to be performed. The program for doing the calculation given above looks like this:

```
10 LET A = 100
20 LET A = A*50 (where * = multiply)
30 LET A = A/25 (where / = divide)
40 PRINT A
```

If you were to insert a print instruction after each operation (that is put an instruction **PRINT A** as Lines 15 and 25) the computer would behave exactly like a calculator in that it would display the answer at each stage of the calculation. However this sort of display is redundant on a computer because the instructions are already entered in the program and once it is running there is nothing to be gained by looking at the partially processed data.

Consider Line 20 in detail. Note first that it is not a conventional equation. A is not equal to A*50 in the usual sense (if it were, 1 would equal 50 which is nonsense). Line 20 is an instruction which says "**LET pigeon hole A** (which currently holds the number 100) **be changed so that it now holds whatever is the result of multiplying A by 50**." In other words, it replaces the contents of pigeon hole A with whatever lies on the right hand side of the equals sign. This is important because it means that the left and right hand sides of an equals sign cannot be swapped over as in conventional algebra. The two instructions below are not the same:

```
10 LET A = B
20 LET B = A
```

In Line 10, the contents of **A** are replaced with whatever is the value of **B**. In Line 20, it is the other way round, **B** is replaced by **A**.

By using 'replacement' to switch from one variable to another, together with the full range of algebraic functions provided by the Basic language, complex calculations can quite easily be set out in the form of a program.

There is one important rule which must be borne in mind. Computers cannot cope with ambiguity. They interpret instructions strictly in accordance with the rules of the computer language being used. It is therefore essential to make absolutely clear what instructions are to be carried out. A human calculator copes with instructions like the two given below by understanding the context in which the formula is to be used:

```
10 LET A = A * B - C

20 LET B = B - C/A
```

Computers cannot understand context. Their interpretation of algebra is governed by something called the "rule of precedence" which tell them precisely how they are to interpret the instructions. The programmer has to make sure that the instructions he writes are in line with these rules. There are two choices (1) to learn the rules of precedence or (2) to use brackets to make absolutely clear what is meant. The bracket solution is much the easiest – it is one thing less to learn at the cost of a bit more typing! The addition of a pair of brackets renders the expressions completely unambiguous:

```
10 LET A = (A*B)-C

OR 20 LET B = B - (C/A)
```

Logic:

In addition to the normal arithmetical functions, computers have facilities for handling logical statements or "truth tables". That is, they can evaluate logical statements and say whether they are "true" of "false". The answer is given by the value assigned to the expression. If it is true, the value is 1 and if it is false, the value is 0. The statement **(D>2)** has the value **1** if **D** is **3** or more and the value **0** if **D** is **2** or less. (> meaning greater than.)

Logic statements can be combined with conventional arithmetic as in the following example:

```
10 LET P = P * (D>2)
```

Here, **P = P * 1** if **D is greater than 2** and **P = P * 0 if D is 2 or less.** Note that the avoidance of ambiguity using brackets is just as important as in straightforward algebraic instructions.

String arithmetic:

String variables can be added in much the same way as numerical variables as in the following example:

```
10 LET A $ = "BOY" : LET B$ = " AND GIRL"

20 LET C$ = A$ + B$

30 PRINT C$
```

The result is boy and girl.

Strings also have their own set of instructions which enable them to be manipulated in ways that have no equivalent in numerical form. The following program strips the initials from a name (provided there are always two initials!).

```
10 LET A$ = "D.J.BLOGGS"

20 LET L = LEN (A$) : REM L IS THE NUMBER OF
CHARACTERS IN A$.

30 LET A$ = RIGHT$ (A$, L-4) : REM A$ IS THE RIGHT (L-
4) CHARACTERS OF A$. REMEMBER THAT FULL
STOPS ARE ALSO CHARACTERS.

40 PRINT A$
```

The result is Bloggs.

Rather surprisingly, one of the most useful applications of string arithmetic is in handling numbers as, for example, when a decimal number has to be turned into an integer.

```
10 LET A = 46.3467
```

```
20 LET A$ = STR$(A) : REM A$ IS THE STRING EQUIVALENT OF 46.3467.

30 LET A$ = RIGHT$ (A$, 2)

40 LET A = VAL(A$)
```

Line 40 converts **A$** back into the numerical variable **A** which has the value **46**.

Arrays:

An array is a systematic list giving the values of a number of variables which are members of the same "family". For example, the sales of a "family" of products could be put into an array in the following way:

Sales of Product 1 = 20
Sales of Product 2 = 30
Sales of Product 3 = 45
Sales of Product 4 = 50

The four numbers which form the column on the right hand side of the equals sign are the array. That is:

20
30
45
50

In Basic, arrays are given names in much the same way as is done with variables – this one is called array **A**. An array is distinguished from other variables by putting the number of elements it contains in brackets after the name. So array **A** is written as **A(4)**. The number in brackets is called the dimension of the array and each element is defined by its "subscript" – that is its number in order from the top of the array. The elements of array **A** are defined as follows:

A(1) = 20

A(2) = 30
A(3) = 45
A(4) = 50

Arrays provide a way of grouping together variables which are going to be subjected to the same operation in a program. Thus instead of laboriously writing out the same instruction for product 1 then product 2 then product 3 and so on, it is only necessary to give one instruction referring to the array as a whole.

As with variables, arrays can be made up of strings as well as numbers but numbers and strings cannot be mixed in the same array. A string array has the distinguishing $ sign after the name, for example **A$(4)**.

Matrices:

A matrix is just a series of arrays put side by side. Matrices are a convenient way of handling arrays which themselves are part of a larger "family". For example, instead of a single array of product sales, there could be a set of sales figures for each of a series of months.

Sales of products 1 to 4 in Jan.
Sales of products 1 to 4 in Feb.
Sales of products 1 to 4 in Mar.

There are 12 numbers in all – that is 4 products multiplied by 3 months. In matrix form this is represented by **A(4,3)**. This represents a "grid" comprising 4 "rows" and 3 "columns" each of which contains a number. (Think of it as an egg box if you like – 4 rows, 3 columns holding a dozen eggs). Sales of product 1 in month 1 (Jan) are given by **A(1,1)**; sales of product 3 in month 2 are given by **A(3,2)**.

As you would expect, a string matrix is written as **A$(,)**.

Loops and branches:

As we have seen, the instructions for getting a computer to do arithmetic are very similar to those used in conventional algebra. The main difference is that the instructions are "strung together" into a program before calculation starts.

This is however only part of the computer programming story. Conventional algebra was invented to match human capabilities and these are not the same as computer capabilities. So it is not surprising that there are programming techniques which do not have a direct equivalent in conventional algebra.

These techniques are all aimed at getting a better return on the effort of writing a program than simply "automating" the instruction sequence used by a human being.

If you think of a computer program as a road map for a journey around the country, the human approach would be to draw up the route so that the number of miles which had to be travelled were minimised. A computer on the other hand would try to maximise the number of times it went over the same bit of road, the reason being that the "cost per mile" (the time taken to do a calculation) for a computer is very very little but the "cost per new set of instructions" (the amount of memory taken up by each new instruction – not to mention the cost of the programmers time) is high.

One result of this is the idea of LOOPS – unknown in conventional algebra but central to computer programming. They come in a number of different forms – the more important of which are described below:

For/next loops:

This type instructs the computer to repeat a calculation for a set number of times. There are two instructions, **FOR** which is at the start of the loop and **NEXT** which is at the finish. Their use is illustrated in the following simple program for printing the numbers Ø to 9:

```
10 FOR N = 0 TO 9

20 PRINT N

30 NEXT N

40 END
```

The loop starts at Line 10 which in effect says "you aren't getting out of this loop until something happens to make the value N greater than 9". Line 20 prints the value of N, which is 0 on the first circuit and Line 30 says "every time you get down to me I will add one to the number represented by N and send you back to the start of the loop at Line 10". This process continues until N has gone from 0 to 9 in steps of 1. When N becomes 10, Line 10 detects that N is no longer within the range 0 to 9 and finishes the loop by a jump to the next statement after **NEXT N,** in this case Line 40, **END**. Note that the value of N which triggers the end of the loop is 10 and not 9. You can check this is you inset a Line 35 PRINT N.

One of the main uses of loops like this is to manipulate matrices and arrays. The example below multiplies each element of array **A(4)** by **2**:

```
10 FOR N = 1 TO 4

20 LET A(N) = A(N)*2

30 NEXT N
```

The same principles can be applied to matrices. The following example multiplies each element of the top row (row 1) in **A(4,3)** by **4**:

```
10 FOR M = 1 TO 3

20 LET A(1,M) = A(1,M)*4

30 NEXT M
```

And to multiply the whole of the matrix by 4:

```
10 FOR N = 1 TO 4

20 FOR M = 1 TO 3

30 LET A(N,M) = A(N,M)*4

40 NEXT M

50 NEXT N
```

This last program contains two "nested" loops – the N loop in Lines 10 and 50 and the M loop in Lines 20 and 40.

Loops must not be allowed to cross. When a **FOR** statement sets up a loop in one variable, the **NEXT** statement which follows must refer to this same variable. The following program has crossed loops and will not work:

```
10 FOR N = 10 TO 20

20 FOR J = 30 TO 40

30 PRINT A(N,J)

40 NEXT N

50 NEXT J
```

The capabilities of a loop are extended by the **STEP** instruction which allows any regular counting sequence to be set up. For example:

```
FOR N = 1 TO 9 STEP2
```

causes N to take the values 1, 3, 5, 7 and then 9, it steps from 1 to 9 with an interval of 2.

```
FOR N = 9 to 1 STEP-2
```

causes N to work from 9 to 1 in steps of -2. i.e. 9, 7, 5, 3, 2, 1.

GOTO:

The **FOR/NEXT** instruction specifies the number of times the program is to go around the loop. **GOTO** can be used to set up a loop in which the exit is controlled by a logical comparison statement. The following program adds blank spaces in order to make a string up to a uniform length of 9 characters:

```
10 INPUT A$: REM A$ CANNOT EXCEED 9 CHARS.

20 IF LEN(A$) = 9 THEN GOTO 50

30 LET A$ = " "+A$

40 GOTO 20

50 PRINT A$
```

The loop is set up in Line 20 which differentiates strings which are less than 9 characters long from those which are already 9 characters long. If they are in the latter group, the program jumps to Line 50 where the string is printed. If the string is less than 9 characters long, a blank space is put in front of the string and it is returned to line 20 to see whether it is now the correct length. If it is still not long enough the process is repeated. When **A$** has been made up to 9 characters, an exit is made from the loop and **A$** is printed at Line 50.

GOSUB . . . . . RETURN:

GOSUB is very much like **GOTO** in that it causes an immediate branch to a specified line. However **GOSUB** goes an extra stage beyond **GOTO**. Once the branch has been made, the computer returns to the original point of departure immediately it encounters a **RETURN** statement. In other words **GOSUB/RETURN** provides a means of looping out of a main program, through a subsidiary program (called a **SUBROUTINE**) and back to the main program. The example below uses a subroutine to create strings of uniform length:

```
10 IF LEN(A$) < 9 THEN GOSUB 1000
```

```
20 PRINT A$

30 END

1000 REM CREATE UNIFORM STRINGS

1010 LET A$ = " "+A$

1020 IF LEN(A$) = 9 THEN GOTO 1040

1030 GOTO 1010

1040 RETURN
```

Lines 10 and 20 are part of the main program. Line 10 branches to line 1000 if A$ is less than 9 characters long. From 1000 to 1030, the string is made up to a length of 9 characters and then Line 10 40 causes a return to the mainstream program at Line 20

IF -- THEN -- GOTO:

The single **IF --- THEN** instruction only handles two states. (e.g. **IF BLACK IS WHITE GOTO 100** covers two cases black is white and black is not white). However, by using more than one such statement, multiple choices can be handled. This is illustrated in the following example where different values of variable R represent the colours of a traffic light. The program converts these numbers into colours.

```
10 IF R=1 THEN GOTO 50

20 IF R=2 THEN GOT0 60

30 IF R=3 THEN GOTO 70

40 GOTO 80: REM PROGRAM ENDS IF R IS NOT 1,2, OR
3.

50 PRINT"RED":GOTO 80

60 PRINT"AMBER": GOTO 80
```

```
70 PRINT"GREEN"
80 END
```

Imagine that R is the outcome of some automatic control system which takes the value of either 1, 2 or 3. Lines 10 to 30 say that depending on the value of R, the program is to branch to one of the specified lines. **R=1** causes a branch to Line 50; **R=2** causes a branch to Line 60 and **R=3** causes a branch to Line 70. At each of these lines the program prints out the appropriate colour. If R is not one of the specified numbers, something is wrong and Line 40 causes a branch to Line 80, **END**.

IF--THEN--GOSUB can be used instead of **IF--THEN--GOTO**. Bear in mind, however, that the program will **RETURN** to the next instruction after the **GOSUB** and this may interfere with the choices which follow.

3

PRINCIPLES of Programming

Introduction:

At the highest level, computer programming is an art form, an elegant blend of mathematics, logic and economy. A good program is like one of those algebra proofs they taught at school. Each stage is a perfect link in a chain of logic. It cannot be improved by adding an extra step and if a single step is removed, the whole thing is invalid. So it is with programming, the aim is to use the minimum number of steps (statements) consistent with there being no logical flaws ("bugs") in any of the program's branches.

Fortunately, this level of perfection is not required just to write a program which works. In straightforward programs, the only price you pay for a lack of elegance is a microscopic decrease in the speed of the calculation and increased use of computer memory which, unless it exceeds what is available, is of no serious consequence.

However, below a certain standard, the whole exercise of writing a program becomes irritating, frustrating and thoroughly unrewarding. There is of course, no official standard but there are many areas where, by exercising a degree of discipline and prior thought, the task of writing a program is greatly eased.

Definition of variables:

It is almost impossible to unravel a program when you are not sure of the definition of each variable. Imagine a program which begins with variable Z being used for the number of days in the month; halfway through Z becomes the sales per day; and at the end Z is a counter in a loop. It can easily happen if you are writing a long program over a lot of evenings. To avoid the problem, the right thing to do is to note each variable definition as a **"REM"** statement at the start of the program. Another approach is to define each variable the first time it appears in the program. If even this cramps your style too much, the absolute minimum is to make a clear distinction between variables used as counters in loops and the rest (i.e. use single letters as counters and double letters as true variables).

Flags:

Flags are variables which are used as indicators of a particular state. Suppose a program is needed to calculate prices in a number of different currencies. The logic of the program is the same, regardless of currency but at certain stages, different constants are applied depending on which currency is being considered. A convenient approach is to set a flag at the start of the program indicating the currency being considered. (That is, a variable **FL** is set to 1 for Pounds, 2 for Marks and so on). When the program reaches a stage where there is a different treatment for each currency, the flag is used to identify which treatment is to be applied. So the program might say **"if FL=1 then price =X whereas if FL=2 then price =1.2 X"**.

In any but the most simple programs, flags need to be kept under strict control. In particular it is very important that they are set back to 0 just as soon as they have finished the task they are intended to perform. Otherwise and particularly if the program goes around a lot of loops and in and out of subroutines, a redundant flag is almost certain to get in the way of the intended logic.

As with counters, it is a good idea to give flags recognisable names – make them 2 characters long and always start with **F** (i.e. **FA, FB, ... FL** etc.)

Subroutines:

The **BASIC** language does not make a big thing out of subroutines. They are only defined by a line number (not a name as in many other languages) and they can go anywhere in the program. Consequently, they can easily be relegated to the status of being just another "chunk" of program which for some vague reason is not part of the mainstream. This is not what subroutines are supposed to be. They should be self-contained blocks of logic, pulled out from the main program because they are used more than once. To enforce this principle, it is a good idea to start each subroutine with a **"REM"** statement which states exactly what it does.

It is generally bad practice to exit from a subroutine part-way through (that is before getting to the **RETURN** statement) (a) because this makes it difficult to follow the logic and (b) because the computer can get into a mess if it does not know that you have finished with the subroutine. It only finds this out when it encounters a **RETURN** statement.

Line numbering:

It is much easier to follow a program if you use a line numbering system which distinguishes different parts of the program from each other by a change in number. For example you could use lines 10 to 100 for variable definitions, lines 500 to 1000 for string definitions, lines 2000 onwards for the main program and start subroutines at 6000, 7000 etc. You can write or buy a program for automatic line re-numbering so you do not have to work out the whole scheme before you start but merely keep things in order as you go along.

A point of detail which can cause a lot of trouble if ignored is that the main program should conclude with an **END** instruction. If it does not, the program will run on into the subroutines and almost certainly produce some unexpected results.

Handling strings:

Except in short, simple programs, it is not a good idea to mix long string statements with the program instructions. It makes the program awkward to follow and gaps tend to appear when the program is edited.

These problems are overcome by using string variables in the program itself and defining them separately either at the start of the program or as they arise. An additional benefit of this approach is that the programmer builds up an inventory of phrases which can sometimes be used on more than one occasion.

Handling numbers:

It is normally necessary to format numbers at the input and output stages of a program. Such operations can become cumbersome if they are done with numerical variables and when this happens you should consider changing over to strings. String arithmetic provides a different set of instructions which often justify the effort involved in converting into and out of strings. For example, when data from a complete calculation has to be displayed as a table, the length of the numbers usually has to be restricted to some maximum number of characters. Overall length can be limited by taking the first **L** characters using a **LEF$(A$)** statement. Another common need is to right justify a column of figures. To do this it is first necessary to convert the number to strings and then find the one which is longest (**L=LEN (A$)**). The rest of the numbers are then made up to the length of the longest by putting blanks in front of them **A$**= " " + **A$**.

Handling matrices and arrays:

Operations on matrices and arrays generally treat all elements in the same way and are usually carried out using loops. The following simple example calculates the % change in sales between two periods.

Two sets of sales figures for 20 products and 10 territories are given by matrix **S(20,10)** for the first period and by matrix **T(,)** for the second period. The % change is yet another matrix **I(,)**. To begin with, consider one element **(1,1)**, which contains sales of product 1 in territory 1. The % change in sales is calculated as follows:

I(1,1) = ((T(1,1) – S(1,1)) * 100) / S(1,1)

To do this same calculation for all 200 product/territory combinations, the formula is encased in two loops which together count their way through the matrices extracting each element in turn:

```
10 FOR M = 1 TO 10

20 FOR N = 1 TO 20

30 LET I(N,M) = ((T(N,M) – S(N,M)) * 100) / S(N,M)

40 NEXT N

50  NEXT M
```

Problems arise in loops if the counters can take positive, zero or negative values in response to values assigned to variables.

Consider the program below which filters out all the numbers over 10 in an array of numbers, **C(N)**, arranged in ascending order.

```
10 INPUT A

20 FOR N = 1 TO 9: A = A+1: C(N) = A: NEXT N

30 FOR N = 1 TO 9

40 IF C(N) > 10 THEN GOTO 60

50 NEXT N

60 PRINT N
```

If there are no numbers less than **N** (i.e. set **A** to **0** in Line 10), the **N** loop counts from **1** to **9** with the result that Line 60 prints **N**, the number of elements less than **10,** as **10**, whereas the answer should be **9**. If there are **5** numbers less than **10**, (i.e. **set A** to **5**), the loop is stopped at Line 40 and the correct answer **N=5** is printed at Line 60.

Next, consider the program below which works out the change in sales, **D( )**, from one month to the next. The period over which the calculation is to be done is defined by **A** and **B**.

```
10 LET S(4) = 16: LET S(3) = 13: LET S(2) = 4: LET S(1) = 1

20 INPUT A: REM START OF SERIES

30 INPUT B: REM END OF SERIES

40 FOR N = A TO B

50 LET D(N) = S(N) - S(N-1)

60 NEXT N

70 FOR N = 1 TO 4: PRINT D(N): NEXT N
```

If **A** takes the value **0 or 1**, the program 'crashes' as it attempts to find a zero or negative element of **S( ).** Thus if months are numbered **1, 2, 3, etc**., the program will not cope with an **A=1** entry.

As will be evident, these simple program can easily be corrected by re-aligning subscripts and line counters. In more complex programs, the danger is that a re-alignment to solve a problem in one section may create a new problem somewhere else.

Chains:

There are occasions when the program's logic dictates that the matrix or array must be operated on in a sequence which cannot be created using a loop. For example, instead of listing an array in sequence from **1 to 6** in steps of 1, it may be necessary to count in a completely irregular sequence such as **6, 2, 5, 4, 3, 1.**

In such a case, an entirely different technique is used in which the counting sequence is defined by another array used as the indexing device. The technique is illustrated in the example below which extracts and prints elements from some previously defined array **Z$(6)** in the irregular sequence given above:

```
10 LET C(6) = 2

20 LET C(2) = 5

30 LET C(5) = 4

40 LET C(4) = 3

50 LET C(3) = 1

60 LET C(1) = 0

70 LET H = 6

80 IF H = 0 THEN GOTO 120

90 PRINT Z$(H)

100 LET H = C(H)

110 GOTO 80

120 END
```

The first time around, Line 90 prints **Z$(6)**. Line 100 then sets **H=C(6)** which has the value of **2**. The next time around, Line 90 prints **Z$(2)**. The circuit is repeated until **Z$(3)** is printed and **H** is set to **C(1)** which is zero in Line 100. This causes the loop to be stopped in Line 80.

Array **C(6)** (which would normally appear in subscript order beginning with **C(1)** and ending with **C(6)**) is called a **chain**. The values of each element of **C** are called **pointers**. The head of the chain is given in Line 60 which sets **H** to **6** and the end of the chain is when **H=0** at **C(1)**.

This idea of pointers can be developed into a very versatile tool for manipulating the order in which matrices and arrays are listed. Its particular virtue is that, instead of actually altering the position of each element using the rather cumbersome nested loop routine described earlier, only the chain array itself needs to be altered if the sequence of the matrix or array is to be rearranged. The primary store of data, that is the data matrix itself, remains the same.

Chains are particularly useful when a number of straightforward and independent sorting operations have to be done. A typical example is the creation of a series of tabulated reports from the same basic data with the data listed in a different order for each report. Instead of sorting the whole matrix, printing it, resorting it, printing it again and so on . . . . . print instructions are given directly by a series of chain arrays which define the order in which the data for each report are to be extracted from the matrix and printed.

User friendliness and idiot proofing:

The most obvious and the important criteria for judging the quality of a program are whether it can be used and whether it is reliable. Or, in the peculiar jargon of computing, whether the program is "user friendly" and "idiot proof".

The following examples illustrate what is meant by these phrases:

– The program asks you to enter some information (the date perhaps). You enter it and the screen goes blank. You wait and nothing happens. What do you do? Wait some more or assume the data you entered was not accepted and start the program all over again. The answer is: you don't know. It is not a user friendly program.

– The program asks you to enter the number of customers in a particular territory. For some reason, you think you should enter the territory name and then the number of customers. You type in N.W. 34, press return. After a short while there is a bleep and an error message appears. What has happened is that the computer expected you to enter a number, you entered a letter, the letter

was accepted but later on, the program "crashed" because it tried to treat a string variable as an arithmetic variable. The program, is not idiot proof.

Virtually all problems of this sort occur when data are being entered into the computer and the only way of avoiding them is to take great care with arrangements for interfacing with the keyboard.

Of course if you are writing programs which you intend only to use yourself, you do not have to be too concerned about how the general public will take to them. You may well think idiot proofing is quite unnecessary since no idiot is going to go near them. But bear in mind that what is obvious when you are writing a program is not always so 6 months or a year afterwards when you come to use it again.

The problem with writing user friendly programs is their size. As you will see, it can take line after line of program to make absolutely sure that all eventualities have been neatly catered for at just one data entry point. Indeed, with the exception of scientific and engineering applications, it is normal for the greater part of a program to be taken up with routines for entering data and for guiding the operator as to what he is being asked to do.

To illustrate the issues involved, consider the problem of assigning a numerical value to the variable **X** where the value of **X** is restricted to an integer number between **1 and 99**.

The simplest approach is to use the **INPUT** command:

```
10 INPUT X
```

and to check that the input has actually taken place, add the lines:

```
20 ↵

30 PRINT X

40 END
```

Type **RUN** and the computer will wait at Line 10 for the value of **X** to be entered. Enter **22** (say) and the program moves on to clear the screen and print the value of **X** – that is 22.

But of course the program is equally happy to accept **220, -220** or **22.22** – it does not know that it is supposed to accept only integer numbers between **1** and **99**. If, however, a letter is entered instead of a number, the program does object. It refuses to accept the entry and prints an error message. The variable **X** has to be equated to a number, not a letter.

The program would be greatly improved if the operator were told what he had to enter. This can be done quite simply by inserting an instruction in Line 10 as follows:

```
10 PRINT "ENTER AN INTEGER NUMBER BETWEEN 1
AND 99";: INPUT A
```

Now typing **RUN** causes the program to print the instruction in Line 10 and then wait for the input as before. The program is rather more user friendly but nothing has been done to make it idiot proof.

As was mentioned earlier, the problem with Line 10 is that if a letter is entered, it is immediately rejected. But that is only part of the problem. The process of rejection does not take place within the program, it takes place in the computer itself (or to be more accurate in the BASIC language program built into the computer). This means that the programmer no longer determines what happens. Control passes to the computer and there is no way of over-riding it. Clearly this is undesirable and such situations must be avoided.

A better approach is to arrange for the computer to accept any entry regardless of whether it is correct or not and sort out the problem entries within the program itself. The first objective can be met by putting the entry into the form of a string. This is done by substituting **X$** for **X** in Line 10. The computer now regards everything as a string and so will accept any keyboard entry without complaint. The second objective of building restrictions into the program itself so that unacceptable entries are rejected, is met by the following two lines:

```
12 IF VAL(X$)<1 or VAL(X$)>99 THEN GOTO 10

14 IF VAL(X$) <> INT (VAL(X$)) THEN GOTO 10
```

Line 12 rejects the entry if its value is not between 1 and 99. This includes entries which are not numbers at all, since **BASIC** evaluates non-numeric strings as 0. Line 14 rejects non-integers by comparing the value of the entry with its integer value. If they are not the same, the entry cannot have been an integer. Rejected entries cause the program to branch back to Line 10 for the operator to re-enter. Finally, since the object is to enter a value for **X, X$** is evaluation and set equal to **X** as follows:

```
16 LET X = VAL(X$)
```

The program cannot now be defeated by idiotic entries. If the entry is not right, it returns to the start and waits for the next one. However, the operator may still be unable to understand why his entries are not being accepted. He needs a prompt to say why he is not getting anywhere. This can be accomplished by the following modification:

```
12 IF VAL (X$)<1 OR VAL(X$)>99 THEN PRINT
"OUTSIDE THE RANGE": GOTO 10

14 IF VAL (X$) <> INT (VAL(X$)) THEN PRINT "NOT AN
INTEGER": GOTO 10
```

Now, if the entry is not accepted, an error message appears to tell the operator why.

The program now meets all the logical requirements but the screen display is quite unworkable. If unaceptable data are entered, the screen becomes a jumble of error messages, input prompts and the entries themselves. The screen display itself must be got under control.

This is done by fixing the location of each print statement with **PRINT** instructions as shown overleaf.

```
5 PRINT "■"
10 PRINT"■■■■■■■ENTER A NUMBER BETWEEN 1
AND 99";
11 INPUT X$
12 IF VAL(X$)<1 OR VAL(X$)>99 THEN PRINT
"■■■■■■■■■■■■■OUTSIDE THE RANGE":GOTO 10
14 IF VAL(X$)<>INT(VAL(X$)) THEN PRINT"■
■■■■■■■■■■■NOT AN INTEGER      ":GOTO 10
16 X=VAL(X$)
20 PRINT"■"
30 PRINT"■■■■■THE ENTRY IS :";X
40 END
```

This is a lot better but still not right. If an entry is rejected and the program goes back to Line 10, the rejected entry remains at the end of the line with the cursor over the first number which was entered – like this:

ENTER A NUMBER BETWEEN 1 AND 99 222

It would be much better if the original entry were removed. To do this, the start of the program must be reorganised so Lines 12 and 14 return to a new Line 10. Line 10 erases any previous entry by overprinting the screen locations where the number appears, with a line of blanks.

There is a similiar problem with the error messages. If a Line 12 error is followed by a Line 14 error, the Line 14 message does not obliterate the Line 12 message. It is O.K. the other way round (i.e. a Line 14 error followed by a Line 12 error) because the Line 12 message is longer than the Line 14 message. One solution is to artificially lengthen the Line 14 message by adding a line of blanks after **INTEGER**.

The full program is now:

```
5 PRINT "□"
10 PRINT"■■■■■■■ENTER A NUMBER BETWEEN 1
AND 99        ■■■■■";
11 INPUT X$
12 IF VAL(X$)<1 OR VAL(X$)>99 THEN PRINT
"■■■■■■■■■■■■■■OUTSIDE THE RANGE":GOTO 10
14 IF VAL(X$)<>INT(VAL(X$)) THEN PRINT"■
■■■■■■■■■■■■NOT AN INTEGER      ":GOTO 10
16 X=VAL(X$)
20 PRINT"□"
30 PRINT"■■■■■■THE ENTRY IS :";X
40 END
```

This is now a reasonably respectable data entry program but note how it has progressively grown from 4 lines to 12. And all it does is enter a number! Anyone could be forgiven for giving up on the whole idea of user friendliness if this sort of effort is needed every time a number has to be entered.

Thankfully, this is not necessary because all data entry routines tend to follow a similar pattern. This makes them ideal candidates for standardising and packaging into subroutines.

Towards a universal data entry subroutine

The **INPUT** subroutines which follow are designed to fulfill the function of a universal data entry subroutine. They work on the following principles:

– Data is requested by a line of text on the screen. For example "Enter your name:".

– The operator can always respond to the request in one of three ways:

1) He can press the **RETURN** key to signify he wants help. He then gets a prompt message telling him what he is supposed to be doing.

2) He can signal an intention to exit from the program by pressing @.

3) He can enter the data which has been requested. The program checks that the input meets the specification and if it does not, it displays an error message and returns to request the information again

Four versions of **INPUT** are listed on pages 46 and 48. Lines 10 to 70 and 300 to 390 are common to all versions. Only Lines 100 to 200 vary. They cater for the four different types of data entry listed below:

– Number Input (**INPUT NUMBERS**)

– Spacebar entry (as in press spacebar to continue) (**INPUT SPACEBAR**)

– Yes or No (**INPUT YES OR NO**)

– Letter input (**INPUT LETTERS**)

These programs are intended to be arranged as subroutines which are called by the main program immediately after a request for data entry has been made. A typical main program segment might be:

```
3022 PRIN"ENTER CUSTOMER NUMBER";
3024 D=6:GOSUB 10:REM SUBROUTINE INPUT S
TARTS AT LINE 10
```

Subroutine **INPUT** receives the data input, checks its validity and returns the data to the main program via **RETURN**.

Back in the main program, the four categories of output (that is **HELP, VALID DATA, INVALID DATA** & **EXIT**) are distinguished from each other by a flag **RF**, called the return flag, which can take the value of either **0**, 1, 2 or 3.

The next lines in the main program use the value of **RF** to sort the four different categories of output and to call up the appropriate response, as follows:

```
3025 IF RF = 1 THEN GOTO 3030

3026 IF RF = 2 THEN GOTO 3032

3027 IF RF = 3 THEN GOTO 3034

3028 REM RF=0 PROVIDES A PROMPT AND THEN
ROUTES BACK TO 3022

3030 REM RF=1 MEANS ENTRY IS VALID – CARRY ON
TO NEXT STAGE

3032 REM RF=2 MEANS AN INVALID ENTRY – EXPLAIN
REASON AND GO BACK TO 3022

3034 REM RF=3 BRANCHES TO EXIT ROUTINE
```

Now consider the subroutines themselves. The purpose of **INPUT NUMBERS** is to enter a series of numbers, including the decimal point and minus sign. In addition, the program provides facilities for restricting the range and types of numbers which are to be admitted.

```
5 REM INPUT DATA ENTRY PROGRAM
10 TV=0:T$=" ":CH=0
20 GET A$:IF A$<>"" THEN GOTO 20
25 GET A$:IF A$="" THEN GOTO 25
30 IF ASC(A$)=13 AND CH=0 THEN RF=0:GOTO
 390
50 IF ASC(A$)=20 THEN GOTO 330
60 IF ASC(A$)=13 THEN RF=1:GOTO 380
70 IF A$="@" AND CH=0 THEN RF=3:GOTO 390
300 GOTO 20
310 T$=T$+A$:PRINT A$;:CH=CH+1
320 GOTO 20
330 IF CH=0 THEN GOTO 20
340 CH=CH-1:PRINT"■■ ■■";
350 IF CH=0 THEN T$="":GOTO 20
360 T$=LEFT$(T$,LEN(T$)-1)
370 GOTO 20
380 TV=CODE(T$)
390 RETURN
```

```
5 REM INPUT NUMBERS
100 IF CH>0 THEN RF=2:GOTO 390
110 IF A$="-" AND CH=0 THEN GOTO 310
120 IF A$="." THEN  GOTO 310
130 IF A$>="0" AND A$<="9" THEN GOTO 310
```

Once the subroutine has been called, the computer waits at Line 20 for a key to be pressed. When this happens, variable **A$** is set to the input character and it is the **ASCII** value of **A$** which determines the action to be taken. Lines 30, 60 and 70 cause a branch out of the main program if **A$** is one of the permitted control characters by which the operator signals either that he wants help,

has finished entering data, or wishes to exit. Line 100 checks that the number of characters entered does not exceed the maximum permitted by variable **D**, set in the main program at Line 3024. This is done by comparing **D** with **CH**, a counter which indexes forward by one each time a character is entered. If **D** is exceeded, an **RF=2** exit occurs via Line 390. Line 110 accepts a minus sign but only if it is the first character. Line 120 accepts a decimal point and finally Line 130 checks that the entry is a number from 0 to 9.

If the entry has still not caused the program to branch, Line 300 loops back to the start of the subroutine at Line 20, so ensuring that the computer does not register any character other than those specified in the program.

If the entry was a valid character it will have branched to Line 310 where **T$** accumulates each entry as it is made and **CH** is advanced by one, before returning to Line 20 to wait for the next entry. When the operator has finished entering the number, he presses **RETURN**. This entry is picked up at Line 60 and causes a branch to Line 380 after setting **RF** to **1**. **TV** is set to the value of the accumulated string **T$** and Line 390 returns to the main program with **RF=1** signifying that a valid entry has taken place.

Now look at what happens if the operator requests a prompt by pressing **RETURN** (alone). Line 30 detects that **A$** is **RETURN** (ASCII code 13) and that it is the first character to be entered (**CH=0**). This causes Flag **RF** to be set to 0 and there is an immediate jumping to **RETURN** at Line 390. This time **RF=0** signifies that a prompt has been requested.

In a similar way, if the operator presses @, Line 70 detects the entry and sets **RF** to 3 before jumping to **RETURN**.

If the operator begins entering data and then chooses to alter it, he presses the arrow key to back space in the normal manner. Line 50 detects the left arrow key entry. Line 330 looks to see whether a character has already been entered and if not (**CH=0**), the entry is ignored. If characters are present, Line 340 carries out a backspace in the following way. First, the value of **CH** is set back one unit. The cursor is then moved back one space and whatever has been printed at that location is erased (**PRINT " ";**). Erasing

the entry causes the cursor to move forward and so it has to be moved back once again. Line 350 caters for the case where all the entries have been erased and stops **CH** going negative. Line 360 removes the latest entry from the accumulator **T$** before returning to get another character via Line 370.

Letter entry is treated in much the same way in **INPUT LETTERS**. In this case the check in Line 110 is set to admit characters **A to Z** but nothing else.

```
5 REM INPUT LETTERS
100 IF CH>0 THEN RF=2:GOTO 390
110 IF ASC(A$)>ASC("A")-1 AND ASC(A$)<AS
C("Z")+1 THEN GOTO 310
```

INPUT SPACE BAR is considerably simpler – there is no need to include **CH** since only one character is entered and the left arrow correction routine is superfluous. They are only kept in to retain consistency.

```
5 REM INPUT SPACE - BAR
100 IF A$=" " AND CH=0 THEN RF=1:GOTO 39
0
```

The last of the subroutines is **INPUT Yes** or **No** listed below.

```
5 REM INPUT YES OR NO
100 IF A$="Y" AND CH=0 THEN T$="1":GOTO
310
110 IF A$="N" AND CH=0 THEN T$="2":GOTO
310
```

Obviously, versions of **INPUT** can be written to cater for any logical restriction on data entry. If integer numbers must be entered, the decimal point can be rejected. If it is considered necessary to restrict the input so only one decimal point can be entered, this can be done by a flag as follows:

```
120 IF A$ = "." AND DF = 0 THEN LET DF = 1: GOTO
310

122 IF A$ = "." AND DF <> 0 THEN GOTO 20
```

It would be quite possible to arrange that the different versions of **INPUT** took the form of subroutines within the main **INPUT** subroutine and call them up using a flag set in the main program. However, unless space is really at a premium it is probably better to keep it simple. Nested subroutines are not all that easy to follow.

Sorting:

The natural way of writing a program is to try and duplicate the way you would solve the problem if you were doing it without a computer, (perhaps with the help of a calculator). As a general rule, there is nothing wrong with this approach — indeed what else are you going to do? But there are occasions where the way you would solve even a very simple problem is just not appropriate for a computer. A good illustration of this is provided by the problem of sorting.

Imagine a list of data comprising the numbers 5,4,6,7,3. which are to be put into a descending order. If you were solving the problem in your head, you would scan the numbers and pick out the largest, then the next largest and so on . . . until all the numbers had been put into the right order. A computer finds that rather difficult to do — it cannot compare more than two things at once. What it has to do therefore, is to compare the first two numbers and find out

which is the bigger; then take the next two numbers and compare them; and so on . . . . . The programmer's task is to harness this limited ability to the task of producing a ranked list.

The way it is done is to use a "**Bubble sort**"; a technique which uses the computers ability to compare two numbers by a sequence which works progressively through the whole list. The sequence begins by taking the first pair of numbers and putting them in size order (that is in a descending sort, it puts the larger number first). In this particular list, the first two numbers (5 and 4) are already in order. The next stage is to discard the larger number and to take the second pair of numbers (that is 4 and 6). Once again they are ranked in order and this time the order is reversed to 6 and then 4. Discarding 6 brings in the next number to get the pair 4 and 7; ranking them changes the order to 7 then 4. The next and last pair in the sequence is 4 and 3 which are already in the right order. This list is now 5,6,7,4,3. — better but not right. Indeed the only one of the list that is right is the last number: 3. If 3 were not the smallest number it could not have ended up at the bottom of the list.

The next stage is to repeat the sort of pairs for the whole list all over again except that this time, the last number (i.e. the smallest which is in the right place) is left off. This produces another ranking where again only the last number is definitely in the right place. If this process is repeated enough times, all but the "top" two numbers will be in the right order — and the computer has no problem, sorting these last two numbers in the right order. In the general case, the number of sorting cycles which have to be gone through is one less than the number of items in the list.

The program shown at the end of the chapter provides a Bubble sorting routine for both ascending and descending orders. It is arranged to handle strings and illustrates how names can be ranked in order by using "**string arithmetic**". Line 420 is where the basic pair comparison takes place. **C$(K)** is compared with **CS (K+1)** to see which is the larger. What actually takes place is a

comparison of the code numbers of the first character of each of the two strings. Since the code assigns numbers to letters in an ascending order from **A** to **Z** (the code for A is 65 and for Z it is 90), the comparison of the two strings identifies their alphabetic order.

The data to be sorted are entered in Line 800, using a **DATA** statement and are then read into the array **C$( )**, where **N** is the number of data items. It would be possible to enter numbers instead of letters but the numbers would first have to be entered as strings (i.e. "1", "2", etc). Note that sorting would still be done by the "string arithmetic" described above so the numbers would be sorted on the value of the first character and only a list of numbers all less than 10 would be correctly sorted in numerical order. To sort numbers properly, **C$( )** must be changed into a numeric array **C( )** and the program rewritten to use numerical variables.

The choice between ascending and descending sorts is made in Lines 300 and 350 and the Bubble sort itself begins at Line 400 where a loop (counter K) is set up to repeat the sorting cycle (N-1) times. The comparison of each pair of entries is carried out in another loop set (or nested) inside the first loop which starts at Line 450. This loop counts "backwards" from K down to 1 in steps of -1. The determinant of whether the sort is to ascend or descend is the way the pair comparison is specified. Line 470 (**C$(J)** >= etc.) ranks in ascending order and Line 490 (**C$(J)**<= etc.) ranks in descending order. The outcome of the comparison of pairs is either that the two numbers are in the right order, in which case the program moves onto look at the next pair of numbers, or that they are in the wrong order and must be "swapped over". This is done in Lines 500 and 520 where **T$** is a temporary store of one of the values. The sort ends when the cycle of comparisons has been completed at Line 550. Finally, the ranked list is printed out and Line 610 signifies that the computation has finished.

```
100 REM SORTING ROUTINE
180 DIM C$(100)
230 N=7
240 REM READ DATA INTO C$ ARRAY
250 FOR J=1 TO N
260 READ M$:C$(J)=M$
270 NEXT J
280 REM DESCENDING OR ASCENDING ORDER
290 PRINT:PRINT:PRINT
300 PRINT"IN WHAT ORDER DO YOU WISH TO S
EARCH ?":INPUT A$
350 IF A$<>"A" AND A$<>"D" THEN GOTO 290
360 REM SORT BEGINS HERE
400 FOR K=1 TO N-1
410 IF A$="A" THEN GOTO 440
420 IF C$(K)>=C$(K+1) THEN GOTO 540
430 IF A$="D" THEN GOTO 450
440 IF C$(K)<=C$(K+1) THEN GOTO 540
450 FOR J=K TO 1 STEP -1
460 IF A$="A" THEN GOTO 490
470 IF C$(J)>=C$(J+1) THEN GOTO 540
480 IF A$="D" THEN GOTO 500
490 IF C$(J)<=C$(J+1) THEN GOTO 540
500 T$=C$(J)
510 C$(J)=C$(J+1)
520 C$(J+1)=T$
530 NEXT J
540 NEXT K
550 REM SORT ENDS
580 FOR L=1 TO N
590 PRINT C$(L)
600 NEXT L
610 PRINT"NORMAL TERMINATION"
800 DATA SIMON,BILL,MARY
810 DATA CAROL,FRED,SUE
820 DATA DAVE
```

4

ADJUSTER
Adjusting a Sales Trend

Introduction:

Figures which show trends in performance over time are always good for raising the temperature of a conversation between business colleagues. To some, a rising sales graph is the just reward for exceptional effort, whilst to others it merely indicates that the customers have started buying again. A sudden dip may herald an anticipated disaster or it may merely reflect that people have not got back from the Christmas break.

This sort of uninformed interpretation of figures does little to aid comprehension. It may be the opinion is right but if so, it owes nothing to the figures under examination. If trends in figures are to be made to give up the information they contain they must be analysed first and discussed later.

A more cynical but no less realistic point of view is summarised by the old adage about lies, damned lies and statistics. If people are going to hold forth with instant opinions as to how well you are doing, it is no bad thing to be prepared with an alternative set of figures which demonstrates there are other interpretations which are equally valid. If the going gets really tough, you may even want to turn your attention to analysing some of the opposition's figures!

A comprehensive discussion of trend analysis is way outside the scope of this book — the subject is a branch of Statistics and has its own specialised literature and computer software. However, there is a lot of mileage to be got from simple analyses of trend data taking account of factors with which everyone is familiar, such as inflation and the effect of the different number of working days in the month. Adjust a sales trend for these two factors and it will almost certainly tell a different story. Things may not look any better than they did before but they will be easier to explain.

ADJUSTER takes a trend of monthly sales figures and adjusts it to take account of either inflation, working days or both inflation and working days.

The idea is that you enter a sales trend of up to 36 monthly figures together with a price index and the number of working days in each month. To adjust the figures for inflation, you convert all the sales figures into what they would have been if prices had remained the same throughout the period. In other words, each sales figure is adjusted by the ratio of "the price ruling at the start of the trend" divided by "the price ruling when the sales were made". Or, in mathematical notation:

ADJUSTED SALES IN MONTH 10 = SALES IN MONTH 10 * (PRICE INDEX IN MONTH 1/PRICE INDEX MONTH 10)

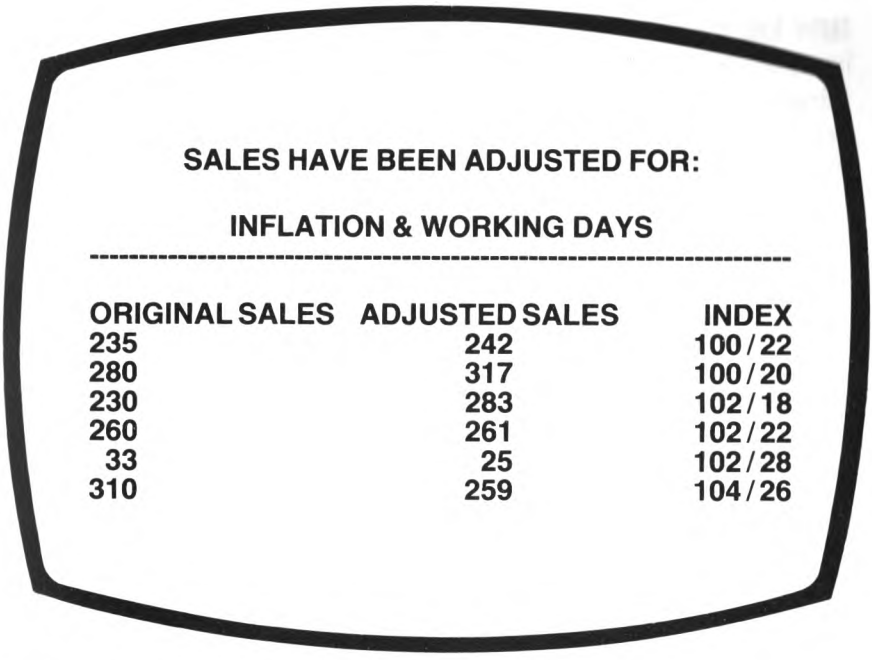
If the price index in month 1 is 100 and the price index in month 10 is 200 then prices have doubled over 10 months. Therefore, to adjust sales in month 10 back to the level they would have been in month 1, you multiply by 100/200.

To adjust for the different number of working days in the month, you work out what sales in the month would have been if the month had contained one twelfth of the number of working days in a year. In other words, the sales are converted to what they would have been if all months were exactly the same length. It is a two stage calculation — the first stage is to calculate the average number of working days per month and the second stage is to multiply each months sales by the ratio "actual number of working days/average number of working days". Again, in mathematical notation:

AVERAGE NUMBER OF WORKING DAYS = SUM OF EACH MONTHS WORKING DAYS/NUMBER OF MONTHS

ADJUSTED SALES IN MONTH = SALES IN MONTH * (ACTUAL WORKING DAYS/AVERAGE WORKING DAYS)

To adjust for both inflation and working days you simply feed the adjusted sales figure from the inflation calculation into the working days calculation in place of "sales in the month".



SALES HAVE BEEN ADJUSTED FOR:

INFLATION & WORKING DAYS

ORIGINAL SALES	ADJUSTED SALES	INDEX
235	242	100/22
280	317	100/20
230	283	102/18
260	261	102/22
33	25	102/28
310	259	104/26

Program description

ADJUSTER incorporates the full procedure for data entry described in Chapter 3 with subroutines 7000, 8000 and 8500 handling the different categories of data entry.

Subroutine 9000 is a standard "help" routine called from the main program in response to an **RF=0**return from one of the data entry subroutines. Subroutine 9500 is a standard "exit" routine called in response to an **RF=3** return. The bottom lines on the screen are reserved for prompt messages. The way the prompt and exit routines work can be seen at the start of the program. Line 1004 asks for the number of months data and calls subroutine 7000 so as to enter a number (**D=2** limits the entry to **2** characters). Lines 1005 to 1007 handle the return from the subroutine and provide three branches depending on the value of **RF. RF=0** "falls through" to Line 1008 where subroutine 9000 is called. This subroutine displays the prompt **B$** (which has just been defined). Note Lines 9002 and 9014 which twice clear the bottom two lines of the screen (a) before the prompt is given (there may already be something written there) and (b) after the prompt has been acted upon.

RF=1 is the exit route for a valid entry (to the extent that the entry is a number and contains 2 characters) and it is routed to Line 10 18 where a further check is made. If the entry fails this check, an error message is printed (Line 1020) and the entry routine is started again. If all is well the data which has been entered is assigned to variable **NM** in Line 1018 before continuing on to the next stage at Line 1102.

RF=3 is the exit routine which calls subroutine 9500 in Line 1012. Subroutine 9500 first checks that the operator really does want to exit. (He may have pressed the wrong key by accident). If an exit is wanted, the program exits in Line 9508. If an exit is not wanted the program routes to **RETURN** and thence back to the start of the data entry routine via Line 1014.

Having established the number of months data which are to be entered, the arrays are dimensioned in Line 1102, ready for the main body of data to be entered in a for/next loop which begins in Line 1104 and ends at Line 1162. The loop counts from **1 to NM** (the number of months) and one month's data are entered at each step of the loop. The three items of data making up an entry (that is, sales, price index and working days) are entered in much the same manner as before, beginning at Line 1108. Sales data are stored twice, first in array **S(M)** and then in **A(M)** (Line 1124). (The reason for this apparent duplication will become evident later). The price index entry starts at Line 1126 and the working days entry starts at Line 1144.

Having entered all the data, the operator is asked to specify which of the three possible analyses are to be carried out. The question is asked and answered in Lines 1200 to 1224. Depending on the answer, flag **AC** is set to be either **1, 2** or **3** and the program branches to the appropriate calculation.

The inflation adjustment calculation is in subroutine 5000 and the working days adjustment is in subroutine 6000. The calculations they perform were described at the start of the chapter.

Note that there are now two sets of sales data to be stored — the "raw" data and the "adjusted" data so another array is needed. This second array is **A(M)** which, for convenience was made equal to **S(M)** in Line 1124.

The data adjustment completed, printing starts at Line 1300. The selection of the correct title is made between Line 1304 and 1312. Line 1314 underlines the title and Line 1316 prints the column headings (which are arranged to be the same regardless of the analysis which has been chosen). Printing is done in a for/next loop beginning at Line 1318 and ending at Line 1330. Up to 36 **(NM)** lines may have to be printed and clearly they will not all fit on to one screen. The listing has therefore to be stopped when the screen fills up and control passed to the operator until he calls for another "screenful" of figures. This exercise is controlled by the two variables **L** and **M** which are set to **1** and **10** respectively in Line 1302. If there are less than 10 readings (that is **10** > **NM)**, **M** is set to the number of readings. Once into the loop at Line 1318, the first 10 lines are printed and when the end of the loop is reached, Line 1332 checks whether all the readings have been printed in the first circuit. If they have not the operator is asked to press the space bar to print another screenful of data. To do this the two variables are indexed forward by 10 in Line 1346 and if the number of readings left to be printed are less than 10, **M** is once again set to **NM**. The print cycle begins again at Line 1304. The process continues until all the readings have been printed. Within the print loop, flag **AC** which takes the value **1**, **2** or **3** depending on the choice made earlier (Line 1322), calls up the correct "adjusted" figures. Line 1328 arranges for both the price index and the number of working days to be printed when **AC**=**3** (the inflation and working days adjustment) has been selected.

When printing is finished, the operator can choose to go through the whole print routine again or exit (Line 1348 onwards). Line 1368 routes either to the exit, Line 9508, or to the start of printing at Line 1302.

```
1000 PRINT"■";
1002 PRINT"■■■■■■■■■■■■■■ADJUSTER":PRINT"
■■■■■■■■■■■■■■==========="
1004 PRINT"■■■■■■■■■■■■■■HOW MANY MONTHS DA
TA ?        ■■■■■■■■";:LET D=2:LET DE=0
1005 GOSUB7000:IF RF=1 THEN GOTO 1018
1006 IF RF=2 OR RF=3 THEN GOTO 1012
1008 LET B$="ENTER INTEGER NUMBER BETWEE
N 1 AND 36":GOSUB 9002
1010 GOTO 1004
1012 GOSUB 9500
1014 GOTO 1004
1018 IF TV>0 AND TV<37 THEN LET NM=TV:GO
TO1102
1020 PRINT"■■■■■■■■■■■■■■■■■■■■■■ENTRY OUTSID
E RANGE":GOTO 1004
1100 REM ENTER SALES, PRICE INDEX AND W.
DAYS
1102 DIM S(NM),P(NM),W(NM),A(NM)
1104 FOR M=1 TO NM
1106 PRINT"■■■■■■■■■ENTER DATA FOR MONTH "
;M
1108 PRINT "■■■■■■■■■■■SALES          ■■■■";:LE
T D=5:LET DE=1:GOSUB 7000
1109 IF RF=1 THEN GOTO 1122
1110 IF RF=2 THEN GOTO 1120
1111 IF RF=3 THEN GOTO 1116
1112 LET B$="ENTER A NUMBER SMALLER THAN
 100000":GOSUB 9002
1114 GOTO 1108
1116 GOSUB 9500
1118 GOTO1106
1120 PRINT"■■■■■■■■■■■■■■■■■■■■■■ENTRY OUTSID
E RANGE":GOTO 1108
1122 PRINT"■■■■■■■■■■■■■■■■■■■■■■
       "
1124 LET S(M)=TV:LET A(M)=TV
1126 PRINT"■■■■■■■■■■■■INDEX          ■■■■";:LE
T D=3:LET DE=0:GOSUB 7000
1127 IF RF=1 THEN GOTO 1138
1128 IF RF=2 THEN GOTO 1142
1129 IF RF=3 THEN GOTO 1134
1130 LET B$="ENTER A NUMBER BETWEEN 100
AND 999":GOSUB 9002
```

```
1132 GOTO 1126
1134 GOSUB 9500
1136 GOTO 1106
1138 PRINT "
                                              "
1140 IF TV>99 AND TV<1000 THEN LET P(M)=
TV:GOTO 1144
1142 PRINT"ENTRY OUTSIDE
 RANGE - TRY AGAIN":GOTO 1126
1144 PRINT"WDAYS          ";:L
ET D=2:GOSUB 7000
1145 IF RF=1 THEN GOTO 1156
1146 IF RF=2 THEN GOTO 1160
1147 IF RF=3 THEN GOTO 1152
1148 LET B$="ENTER A NUMBER BETWEEN 10 A
ND 31":GOSUB 9002
1150 GOTO 1144
1152 GOSUB 9502
1154 GOTO 1106
1156 PRINT "
                                              "
1158 IF TV>9 AND TV<32 THEN LET W(M)=TV:
GOTO 1162
1160 PRINT "ENTRY OUTSI
DE RANGE - TRY AGAIN":GOTO 1144
1162 NEXT M
1200 REM CHOOSE WHICH ADJUSTMENT TO MAKE
1202 PRINT"ENTER HOW DATA ARE TO
BE ADJUSTED"
1204 PRINT"1) INFLATION"
1206 PRINT"2) WORKING DAYS"
1208 PRINT"3) INFLATION AND WORKING DAY
S"
1210 PRINT"ENTER THE APPR
OPRIATE NUMBER";:LETD=1:GOSUB 7000
1211 IF RF=1 THEN GOTO 1222
1212 IF RF=2 THEN GOTO 1224
1213 IF RF=3 THEN GOTO 1218
1214 LET B$="ENTER A NUMBER BETWEEN 1 AN
D 3":GOSUB 9002
1216 GOTO 1202
1218 GOSUB 9502
1220 GOTO 1202
```

```
1222 IF TV>0 AND TV<4 THEN LET AC=TV:GOT
O 1226
1224 PRINT"■■■■■■■■■■■■■■■■■■■■ENTRY OUTSID
E RANGE - TRY AGAIN":GOTO 1202
1226 REM MAKE ADJUSTMENTS
1227 IF AC=1 OR AC=3 THEN GOSUB 5000
1228 IF AC=2 THEN GOSUB 6000
1230 IF AC<>3 THEN GOTO 1302
1232 GOSUB 6000
1300 REM PRINT OUTPUT
1302 LET L=1:LET M=10:IF M>NM THEN LET M
=NM
1304 PRINT"■SALES HAVE BEEN ADJUSTED FOR
 "
1305 IF AC=1 THEN GOTO 1308
1306 IF AC=2 THEN GOTO 1310
1307 IF AC=3 THEN GOTO 1312
1308 PRINT"INFLATION":GOTO1314
1310 PRINT"WORKING DAYS":GOTO 1314
1312 PRINT"INFLATION AND WORKING DAYS"
1314 PRINT"=========================="
1316 PRINT"ORIG.SALES ADJ.SALES        IN
DEX"
1318 PRINT:FOR N=L TO M
1320 PRINT S(N),A(N),;
1322 IF AC=2 THEN GOTO 1326
1323 IF AC=3 THEN GOTO 1328
1324 PRINT P(N):GOTO 1330
1326 PRINT W(N):GOTO 1330
1328 PRINT P(N);" / ";W(N)
1330 NEXT N
1332 IF M=NM THEN GOTO 1348
1334 PRINT"■■PRESS SPACE BAR TO CONTINUE
":GOSUB 8000
1346 LET L=L+10:LET M=M+10:IF M>NM THEN
LET M=NM
1347 GOTO 1304
1348 PRINT"■■■■■■■■■■■■■■■■■■■■DO YOU WAN
T TO :"
1349 PRINT"    1) LOOK AT THE TABLE
               2) EXIT"
1350 LET D=1:GOSUB 7000
1351 IF RF=1 THEN GOTO 1362
1352 IF RF=2 THEN GOTO 1366
1353 IF RF=3 THEN GOTO 1358
1354 LET B$="ENTER 1 TO START LISTING AG
```

```
AIN":GOSUB 9002
1356 GOTO 1348
1358 GOSUB 9502
1360 GOTO 1348
1362 PRINT"SQQQQQQQQQQQQQQQQ]
                                   "
1364 IF TV>0 AND TV<3 THEN GOTO 1367
1366 PRINT "SQQQQQQQQQQQQQQQQ]ENTRY OUTSI
DE RANGE - TRY AGAIN":GOTO 1348
1367 IF TV=1 THEN GOTO 1302
1368 IF TV=2 THEN GOTO 9508
1370 END
5000 REM INFLATION ADJUSTMENT
5002 FOR M=1 TO NM
5004 LET A(M)=A(M)*(P(1)/P(M))
5005 LET A(M)=INT(A(M))
5006 NEXT M
5008 RETURN
6000 REM WORKING DAYS ADJUSTMENT
6002 LET TD=0
6004 FOR M=1 TO NM
6006 LET TD=TD+W(M)
6008 NEXT M
6010 FOR M=1 TO NM
6012 LET AD=TD/NM
6014 LET A(M)=A(M)*(AD/W(M))
6016 LET A(M)=INT(A(M))
6018 RETURN
7000 REM NUMBER INPUT
7002 LET TV=0:LET T$=" ":LET CH=0
7004 GET A$:IF A$<>"" THEN GOTO 7004
7005 GET A$:IF A$=""THEN GOTO 7005
7006 IF ASC(A$)=13 AND CH=0 THEN LET RF=
0:GOTO 7042
7010 IF ASC(A$)=20 THEN GOTO 7030
7012 IF ASC(A$)=13 THEN LET RF=1:GOTO 70
40
7014 IF ASC(A$)=95 AND CH=0 THEN LET RF=
3:GOTO 7042
7016 IF CH=D THEN LET RF=2:GOTO 7042
7018 IF A$="E" THEN LET RF=4:GOTO 7042
7020 IF A$>="0" AND A$<="9" THEN GOTO 70
26
7022 IF DE=1 AND A$="." THEN GOTO 7026
7024 GOTO 7004
7026 LET T$=T$+A$:PRINT A$;:LET CH=CH+1
```

```
7028 GOTO 7004
7030 IF CH=0 THEN GOTO 7004
7032 LET CH=CH-1:PRINT "■■ ■■";
7034 IF CH=0 THEN LET T$="":GOTO 7004
7036 LET T$=LEFT$(T$,LEN(T$)-1)
7038 GOTO 7004
7040 LET TV=VAL(T$)
7042 LET D=0:RETURN
8000 REM SPACE BAR INPUT
8004 GET A$:IF A$=CHR$(32) THEN GOTO 800
4
8006 GET A$:IF A$<>CHR$(32) THEN GOTO 80
06
8034 TV=1:RF=1:RETURN
8500 REM YES OR NO INPUT
8502 LET RF=0:LET CH=0
8504 GET A$:IF A$<>"" THEN GOTO 8504
8505 GET A$:IF A$="" THEN GOTO 8505
8506 IF ASC(A$)=95 THEN LET RF=3:GOTO 85
14
8508 IF A$="Y" THEN LET RF=1:LET TV=1:GO
TO 8514
8510 IF A$="N" THEN LET RF=1:LET TV=2:GO
TO 8514
8512 GOTO 8504
8514 PRINT A$:RETURN
9000 PRINT"■■■■■■■■■■■■■■■■■■■■
                                "
9001 PRINT"
            "
9002 PRINT"
            "
9004 PRINT"■■■■■■■■■■■■■■■■■■■■";B$
9005 PRINT"PRESS SPACE BAR TO CONTINUE":
GOSUB 8000
9006 IF RF=1 THEN GOTO 9014
9008 GOTO 9002
9010 GOSUB 9502
9012 GOTO9002
9014 PRINT "■■■■■■■■■■■■■■■■■■■■
                                "
9015 PRINT"
            "
9016 PRINT"
            "
9017 RETURN
```

```
9500 REM ESCAPE ROUTINE
9502 PRINT"■■■■■■■■■■■■■■■■■■■■DO YOU WANT T
O EXIT (Y/N) ?":GOSUB 8500
9503 IF RF=1 THEN GOTO 9506
9504 IF RF=2 OR RF=3 THEN GOTO 9502
9506 IF TV=2 THEN GOTO 9510
9508 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■THE END■■
■■■■■■■■=========":END
9510 PRINT"■":RETURN
```

In conclusion:

When the program is running, take a look at the following 12 months worth of data. What sort of year does it look like? Is it getting better or worse? Is it seasonal? Has the rapid growth of the first quarter run out of steam for the rest of the year?

Month	Sales	Price Index	Working days
Jan	83	100	16
Feb	94	100	18
Mar	109	100	21
Apr	106	102	20
May	95	102	18
Jun	106	102	20
July	114	104	21
Aug	97	104	18
Sept	114	104	21
Oct	115	106	21
Nov	110	106	20
Dec	94	106	17

Now run the data through **ADJUSTER.**

The program would be greatly improved by the inclusion of a facility for storing the data on tape or on disc at the end of the

program. A further refinement would be a facility for adding an extra month's data onto the data file so that a monthly trend analysis could be done on an on going basis, keying the information in only once. Ways of adding these features will become apparent from the chapters which follow.

5

GRAPHPLOTTER
Plotting Graphs and Charts

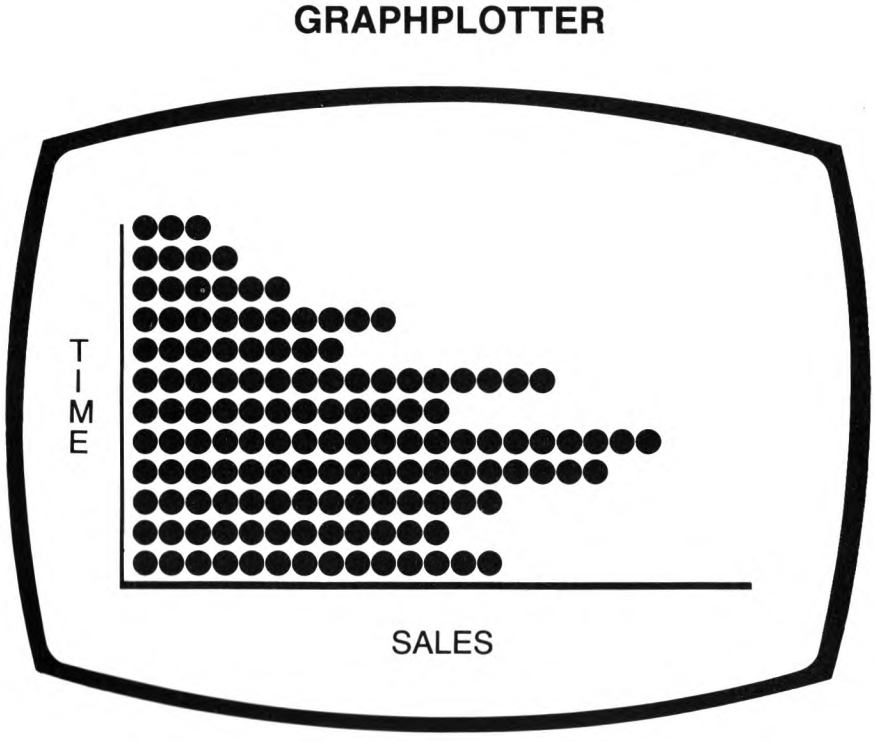
Introduction:

Ingenious screen displays have become the hallmark of personal computing. Indeed the standard of display on a personal computer can be considerably higher than that achieved by some commercial "mainframe" programs. Graphics are particularly important in personal computer programs because of the interactive nature of the tasks they perform and their reliance on screen output as opposed to printing.

At least in principle, graphics programming is a very straightforward business — all you have to do is to join points on the screen with lines. However, in practice, you need to know something about how a computer draws graphics and the features of your particular machine before tackling graphics programming.

Program description:

To illustrate what **GRAPHPLOTTER** does, imagine a bar chart of 12 month sales data. The chart comprises 12 horizontal lines with each line representing sales in a particular month. The horizontal axis is scaled from Ø to the maximum value of monthly sales (or perhaps from the minimum sales value to the maximum value). The vertical (TIME) axis is arranged so the earliest month is at the top of the screen and subsequent months appear on adjacent lines below.



GRAPHPLOTTER starts by entering the data from which the graph is to be plotted. Since there are a lot of figures to be entered, a relatively sophisticated data entry routine is provided. Its main feature is that is allows errors to be corrected without having to start data entry all over again.

Lines 1000 to 1026 enter the number of data points (that is the number of months), in the same way as was described in earlier chapters.

The main data entry routine is designed so that data are entered at the bottom of the screen and the numbers scroll from bottom to top. Thus each new entry goes "underneath" the previous one. If an error is made, the operator can edit "then and there" by pressing the **E** key. The routine starts at line 1032 where a loop in **J** is set up to count from **1 to NM** (the number of entries). On the first entry (**J=1**), the program jumps to Line 1042 and the entry is assigned to **D( )** at Line 1066. The data are printed on screen line 20 (see program Line 1044). Data entry is controlled by the familiar subroutines described in Chapter 3 except that in this program, **RF** can take one of four values; **RF=4 being the E** (edit) key. Flag **FY is zero**. (When it becomes equal to 1 at a later stage, this signifies the checking routine has been invoked). Assuming a valid entry, the **J=1 to NM** loop indexes forward at Line 1068 and the program starts again at Line 1032.

Imagine now that **D( )** contains a number of entries in which case **J** is no longer equal to 1 and Lines 1036 to 1040 come into play. These lines list the entries already stored in array **D( )**. The listing is so arranged that the latest entry appears at screen line 19 — that is one line above the data entry line. The effect of this is to produce a "column" of entries with the first entry at the top of the column and the last entry just above the data entry line. The new entry is printed on line 20 and so takes its place at the bottom of the

column. As data are entered, the height of the "column" of numbers grows until it reaches the top of the screen. Line 1Ø36 in conjunction with 1Ø38 ensures that when this occurs, only that part of the "column" which will fit on the screen is listed. Thus when **J=2Ø**, **N** runs from **1 to 19** and the screen is just filled with lines of data. When **J** is greater than 2Ø, **N** goes to **zero** but zero values are rejected in Line 1Ø38 with the effect that, after **J** exceeds **2Ø**, the loop always runs from 1 to 19 and the number of entries appearing on the screen stays within capacity.

The **EDIT** routine resides in subroutine 6ØØØ which is called by an **E** keyboard entry via Line 1Ø72. The aim of the subroutine is to set up an "auxiliary" data entry routine. **JJ** is set equal to **J** and 'remembers' how many data points had been entered at the time **E** was pressed. Lines 6ØØØ to 6ØØ8 locate the cursor alongside the top of the "column" of numbers. The **RF** exits from subroutine 7ØØ Ø are used somewhat differently here. **RF=Ø (RETURN** on its own) signals that the existing entry is correct and does not need editing. If **RF** is **zero**, the cursor is moved down one position by reducing counter **JJ** by one in Line 6Ø12. (Note that when **JJ** gets to **1**, the edit routine has to finish because the cursor has descended to the data entry line). If a correction is made, it appears as an **RF=1** entry and the value of the appropriate element of array **D( )** is set to the new value of **TV** in Line 6Ø18. Line 6Ø12 repositions the cursor.

With data entry complete, the next stage is to offer the operator an opportunity to check over the data. This is done by going through the data entry routine again but this time with flag **FY** set to the value **1**. **FY** is set to **1** in Line 2Ø14 in response to the question "do your want to correct the entries?" in Line 1Ø8Ø.

The checking routine uses the same screen display as the data entry routine. However, instead of waiting at the end of the prompt line for data to be entered, in the checking mode the program prints out the entry which has already been made. The operation is controlled by flag **FY**, which equals **1**. **FY** first becomes effective in Line 1Ø46 where it causes the 7ØØØ subroutine to be entered not at its start but part way through at Line 7ØØ4. Values for **CH** and **T$**, derived from the data being held in **D( )** are fed into the subroutine. The effect is to "fool" it into believing that a data entry is already in progress. As a result, the subroutine prints the value of **T$** and places the cursor immediately ahead waiting for the next entry of **A$** (Line 7ØØ4). If no change is necessary, the operator immediately presses return and exits from subroutine 7ØØØ with **RF** at **1** and **TV** taking the value it was given in Line 146 — that is the original value of **D( )**. If a change is needed, subroutine 7ØØØ behaves just as it does during normal data entry. The operator backspaces, puts in new data, presses **RETURN** and exits from subroutine 7ØØØ with **RF** at **1** and **TV** containing the revised entry. This is then transferred to **D( )** in Line 1Ø66. Note the remote possibility of an **RF=Ø** exit which is handled like a **RETURN** only entry by Line 1Ø53.

With data entry concluded, the next stage is to start working out the shape of the graph. This follows much the same procedure as you would adopt if you were doing the job by hand. Maximum and minimum values of sales **(Y)** are calculated by the first stage of a bubble sort in Lines 252Ø to 255Ø and are displayed for the operator to choose how he wants the graph to be plotted. The operator is not allowed to choose values for the sales axis which lie inside the maximum to minimum of the data since otherwise there is a possibility of the graph "going off the screen". (Line 2584 and 26Ø4).

The next stage of setting out the graph begins at Line 3000 where the screen position of the line representing the sales axis is calculated. The objective is to arrange the scale so that the axis runs from Y max to Y min with a maximum of 10 subdivisions along its length. In addition, the interval between the divisions must be a whole number. It will be apparent that "something has to give" if all these requirements are to be met. The "something" is the upper value of the axis (that is Y max). The calculation stages are set out below. They appear much more complicated than they really are. It is almost exactly the same procedure as you follow when drawing a graph by hand.

1. Calculate (Y max. - Y min.) as specified by the operator. (Line 3002).

2. If (Y max. - Y min.) is more than 10 units, divide the scale by 5. (Line 3006). If that does not produce an integer number then go back and divide the length by 2. If that still does not produce an integer, add 1 to Y max and repeat the process (Line 3010). As an example, suppose the operator set Y -max to 12 and Y -min to 1. (Y max. - Y min.) is 11, which is too big. First 11 is divided by 5 but this does not produce an integer. Next, it is divided by 2 but once again 11/2 is not an integer. Y-max is then increased by 1 to 12 and the process is repeated. This time 12 is divisible by 2 and as 6 is less than 11 (Line 3004), this becomes the value of ND (the number of scale divisions).

3. The position of Y min (that is the origin of the graph) is fixed at the left hand side of the screen. This gives a maximum of 40 plotting points across the screen, available for the horizontal sales line. Line 3014 calculates the number of pixels per division (which must be an integer number). Line 3018 calculates the scale in terms of the number of pixels per unit of sales and so fixes the position of the other end of the line.

4. The bar chart can accommodate up to 24 data points (this was set in Line 1006). Each data point appears as a line (of dots), the first occupying the top row of the screen, the second occupying the second row of the screen and so on ....

Having established the dimensions of the graph in terms of plot points, the lines themselves are drawn on the screen. Line 5004 sets up a counter (**N**) which controls the vertical spacing and Line 5006 sets up another counter (**L**) which controls the length of the line of dots representing sales.

Finally, Line 5020 tells the operator he can now either exit or start the program again and Line 5032 puts his choice into effect.

```
1000 REM GRAPHPLOTTER
1001 CD$="
"
1002 FY=0
1004 PRINT"GRAPHPLOTTER":PRINT"========
====="
1006 PRINT"ENTER NUMBER OF DATA POINTS
, AN INTEGERBETWEEN 2 AND 24"
1008 PRINT"
";:D=4:GOSUB 7000
1009 IF RF=1 OR RF=2 THEN GOTO 1020
1010 IF RF=3 THEN GOTO 1016
1012 B$="INTEGER NUMBER BETWEEN 2 AND 24
":GOSUB 9000
1014 GOTO 1004
1016 GOSUB 9500
1018 GOTO 1004
1020 IF TV<25 AND TV>2 THEN GOTO 1024
1022 PRINT"INTEGER NUMB
ER BETWEEN 2 AND 24 ":GOTO 1008
1024 PRINT"
                    "
1026 NM=TV:DIM D(NM)
1028 PRINT"DATA ENTRY"
1030 PRINT"PRESS 'E' TO EDIT"
1032 FOR J=1 TO NM
1034 IF J=1 THEN GOTO 1042
1036 FOR N=21-J TO 19
```

```
1038 IF N>0 THEN PRINT"█";LEFT$(CD$,N);"
█████████████████           ███████";
1039 PRINT D(J-(20-N))
1040 NEXT N
1042 PRINT"█";LEFT$(CD$,20);"█";
1044 PRINT"ENTER NO. ";J;"....":PRINT"█"
;LEFT$(CD$,20);"██████████████████";
1045 PRINT"       ███████";
1046 IF FY>0 THEN T$=STR$(D(J)):CH=LEN(T
$):PRINT T$;:D=6:GOSUB 7004:GOTO 1050
1048 D=6:GOSUB 7000
1049 IF RF=1 THEN GOTO 1064
1050 IF RF=2 THEN GOTO 1062
1051 IF RF=3 THEN GOTO 1058
1052 IF RF=4 THEN GOTO 1072
1053 IF FY=1 THEN GOTO 1064
1054 B$="ENTER VALUE OF Y CO-ORDINATE":G
OSUB 9000
1056 GOTO 1042
1058 GOSUB 9500
1060 GOTO 1042
1062 PRINT"█████████████████████NUMBER GREAT
ER THAN SIX CHARS":GOTO 1042
1064 PRINT"█████████████████████
                    "
1066 D(J)=TV
1068 NEXT J
1070 GOTO 1076
1072 GOSUB 6000
1074 GOTO 1034
1076 IF FY=1 THEN FY=0
1078 PRINT"█"
1080 PRINT"DO YOU WANT TO CORRECT THE EN
TRIES ?";
1082 GOSUB 8500
1083 IF RF=1 OR RF=2 THEN GOTO 1094
1084 IF RF=3 THEN GOTO 1090
1086 B$="ENTER Y OR N":GOSUB 9000
```

```
1088 GOTO 1076
1090 GOSUB 9500
1092 OGOTO 1076
1094 IF TV=1 THEN GOTO 2000
1096 IF TV=2 THEN GOTO 2500
2000 REM CHECKING ROUTINE
2002 PRINT"■"
2004 PRINT"CHECK ON ENTRY":PRINT"=======
========"
2006 PRINT"IF O.K. THEN RETURN"
2008 PRINT"IF WRONG THEN REENTER"
2010 PRINT"PRESS 'E' TO EDIT"
2014 FY=1:GOTO 1032
2500 REM CALCULATE YM AND YN
2510 YM=0:YN=D(1)
2520 FOR N=1 TO NM
2530 IF D(N)>YM THEN YM=D(N)
2540 IF D(N)<YN THEN YN=D(N)
2550 NEXT N
2558 PRINT"■"
2560 PRINT"■ENTER THE MIN AND MAX VALUES
 FOR THE Y AXIS"
2562 PRINT"■■THE LARGEST ENTRY IS ";YM
2564 PRINT"■■THE SMALLEST ENTRY IS ";YN
2566 PRINT"■■ENTER Y MAX          ■■■■■■■■
■■";
2568 D=8:GOSUB 7000
2569 IF RF=1 THEN GOTO 2582
2570 IF RF=2 THEN GOTO 2580
2571 IF RF=3 THEN GOTO 2576
2572 B$="THIS IS TO SET Y AXIS MAXIMUM A
ND        MINIMUM":GOSUB 9000
2574 GOTO 2566
2576 GOSUB 9500
2578 GOTO 2566
2580 PRINT"■■■■■■■■■■■■■■■■■■■■NUMBER GREAT
ER THAN 8 CHARS";:GOTO 2566
2582 PRINT"■■■■■■■■■■■■■■■■■■■■
```

```
                "
2584 AM=TV:IF AM<YM THEN PRINT"■■■■■■■■■■
■■■■■■■■Y MAX < LARGEST ENTRY":GOTO 2560
2586 PRINT"■■■■■■■■■■■■ENTER Y MIN
  ■■■■■■■■■";
2588 D=8:GOSUB 7000
2589 IF RF=1 THEN GOTO 2602
2590 IF RF=2 THEN GOTO 2600
2591 IF RF=3 THEN GOTO 2596
2592 B$="THIS IS TO SET Y AXIS MAXIMUM A
ND        MINIMUM":GOSUB 9000
2594 GOTO 2586
2596 GOSUB 9500
2598 GOTO 2586
2600 PRINT"■■■■■■■■■■■■■■■■■■NUMBER GREATE
R THAN 8 CHARS";:GOTO 2586
2602 PRINT"■■■■■■■■■■■■■■■■■■
                "
3000 REM CALCULATE NUMBER OF DIVISIONS I
N Y AXIS
3002 ND=AM-AN
3004 IF ND<11 THEN GOTO 3014
3006 ND=ND/5:IF (ND-INT(ND))<0.001 THEN
GOTO 3004
3008 ND=ND*5/2:IF (ND-INT(ND))<0.001 THE
N GOTO 3004
3010 AM=AM+1
3012 GOTO 3002
3014 YI=INT(40/ND)
3016 XI=INT(24/(NM-1))
3018 SC=(YI*ND)/(AM-AN)
5000 REM PLOT POINTS
5002 PRINT"■"
5004 FOR N=1 TO NM
5006 FOR L=1 TO D(N)*SC
5008 PRINT"●";
5010 NEXT L
5012 PRINT:NEXT N
5014 PRINT"PRESS 1 TO START AGAIN OR 2 T
O EXIT";
```

```
5016 D=1:GOSUB 7000
5021 IF RF=1 THEN GOTO 5032
5023 IF RF=3 THEN GOTO 5028
5026 GOTO 5020
5028 GOSUB 9500
5030 GOTO 5002
5032 IF TV=1 THEN RUN
5033 IF TV=2 THEN GOTO 5028
6000 REM EDIT ROUTINE
6002 JJ=J:IF JJ>18 THEN JJ=18
6004 PRINT"■";LEFT$(CD$,20-JJ);"■■■■■■■■■
■■■■■■■■■■■■■■CORRECTION"
6006 REM '*' IS ARTIFICIAL CURSOR
6008 PRINT"■";LEFT$(CD$,21-JJ);"■■■■■■■■■
■■■■■■■■■■■■■■*■■";:D=6:GOSUB 7000
6009 IF RF=1 THEN GOTO 6016
6010 IF RF=2 THEN GOTO 6022
6011 IF RF=3 THEN GOTO 8300
6012 PRINT"■";LEFT$(CD$,21-JJ);"■■■■■■■■■
■■■■■■■■■■■■■■ ";
6013 JJ=JJ-1:IF JJ<=1 THEN GOTO 6024
6014 GOTO 6008
6016 REM
6018 D(J-JJ+1)=TV:GOTO 6013
6020 REM
6022 PRINT"■■■■■■■■■■■■■■■■■■■■■■IF WRONG THE
N REENTER":GOTO 6008
6024 RETURN
7000 REM NUMBER INPUT
7002 TV=0:T$=" ":CH=0
7004 GET A$:IF A$<>"" THEN GOTO 7004
7005 GET A$:IF A$="" THEN GOTO 7005
7006 IF ASC(A$)=13 AND CH=0 THEN RF=0:GO
TO 7040
7010 IF ASC(A$)=20 THEN GOTO 7028
7012 IF ASC(A$)=13 THEN RF=1:GOTO 7038
7014 IF A$="@" THEN RF=3:GOTO 7040
7016 IF CH=D THEN RF=2:GOTO 7040
7018 IF A$="E" THEN RF=4:GOTO 7040
```

```
7020 IF A$>="0" AND A$<="9" THEN GOTO 70
24
7022 GOTO 7004
7024 T$=T$+A$:PRINT A$;:CH=CH+1
7026 GOTO 7004
7028 IF CH=0 THEN GOTO 7004
7030 CH=CH-1:PRINT"█ █";
7032 IF CH=0 THEN T$="":GOTO 7004
7034 T$=LEFT$(T$,LEN(T$)-1)
7036 GOTO 7004
7038 TV=VAL(T$)
7040 D=0:RETURN
8000 REM SPACE BAR INPUT
8002 TV=0:T$=" ":CH=0
8004 GET A$:IF A$=" " THEN GOTO 8004
8005 GET A$:IF A$<>" " THEN GOTO 8005
8034 RF=1:TV=1:RETURN
8500 REM YES OR NO INPUT
8502 TV=0:T$=" ":CH=0
8504 GET A$:IF A$<>"" THEN GOTO 8504
8505 GET A$:IF A$="" THEN GOTO 8505
8506 IF A$="@" THEN RF=3:GOTO 8514
8508 IF A$="Y" THEN RF=1:TV=1:GOTO 8514
8510 IF A$="N" THEN RF=1:TV=2:GOTO 8514
8512 GOTO 8504
8514 PRINT A$:RETURN
9000 REM HELP ROUTINE
9001 PRINT"SQQQQQQQQQQQQQQQQQQQQQ
                                  ";
9002 PRINT"
          ";
9003 PRINT"
          ";
9004 PRINT"SQQQQQQQQQQQQQQQQQQQQQ";B$:PRINT"P
RESS SPACE BAR TO CONTINUE":GOSUB 8000
9005 IF RF=1 THEN GOTO 9014
9006 IF RF=2 THEN GOTO 9008
9007 IF RF=3 THEN GOTO 9010
9008 GOTO 9002
```

```
9010 GOSUB 9502
9012 GOTO 9002
9014 PRINT"
                                        ";
9015 PRINT"
          ";
9016 PRINT"
          ";
9018 RETURN
9500 REM ESCAPE ROUTINE
9502 PRINT"DO YOU REALL
Y WANT TO EXIT ?";:GOSUB 8500
9503 IF RF=1 THEN GOTO 9506
9504 IF RF=2 OR RF=3 THEN GOTO 9502
9506 IF TV=2 THEN GOTO 9510
9508 PRINT"THE END
=========":END
9510 PRINT"":RETURN
```

6

FORECASTER
Sales Forecasting

Introduction:

Sales forecasting is a vitally important aspect of sales management. It requires knowledge, judgment, skill and luck if the outcome is to be in line with the forecast. Computers can only make a small contribution to the total task of preparing a considered forecast but, nonetheless, the contribution can be of considerable help. There is one proviso, the forecaster must understand what the computer is doing.

6

FORECASTER
Sales Forecasting

FORECASTER uses an exponential smoothing model to extend the trend exhibited by a set of data on into the future. The forecasting model requires at least six periods of actual data from which to calculate the trend and it produces a forecast for six periods into the future.

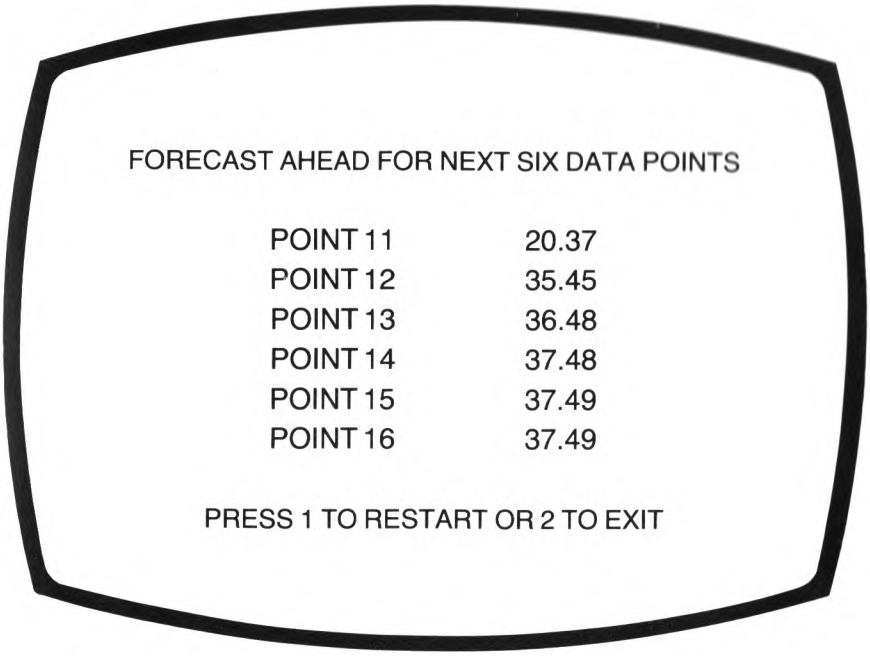
Actual data are entered, a forecast is calculated and forecast sales are displayed as a table. The reliability of the forecast can be improved by feeding "raw" sales data into **ADJUSTER** to eliminate the effects of inflation and working days before using them as the basis for a forecast.

The forecasting model.

The model requires that a minimum of six months past data is available. To make a forecast, it progresses from the oldest data to the newest data in period by period stages.

Each period, the model estimates future sales and compares the estimate with what actually happened. If the forecast is different from the actual, the model uses the error to adjust its forecast for the next period. This process of forecasting, comparing the result with what actually happened, and then adjusting is carried out at least 5 times.

FORECASTER



When complete, the model has worked itself up to the most recent period from where it projects forward to give a forecast of the next six periods sales.

A simple example illustrates the main ideas of this process of forecasting, adjusting and projecting forward.

Assume that monthly sales for, say, February have been forecast to be 200 units. Suppose that actual sales in February turn out to be 220 units, 20 units higher than forecast. (This discrepancy is the error referred to above.) Having observed sales higher than forecast, the old forecast must now be adjusted to give a forecast for March.

This is done by taking the old forecast and adding to it some fraction of the error, say 1/2 of the error. (Notice that if the forecast had exceeded actual sales, the error would have been negative and the effect would have been to lower the new forecast). The resulting new forecast is 200 + 1/2 * 20 = 200 + 10 = 210 units and it is this figure which is projected forward as the sales in March.

The table below shows the main steps:

MONTH	ACTUAL SALES	F'CAST SALES	ERROR	NEW F'CAST
Feb.	220	200	20	210
Mar.		210		

The second table shows how a complete forecast is made using 6 months data covering the period from January to June. Notice that to start the process off, the forecast for the second month (in this case February) is assumed to be the same as the first month (January) – there being no basis for forecasting anything else with only one month's data.

Jan.	200	-	-	-
Feb.	220	200	20	210
Mar.	208	210	-2	209
April	191	209	18	218
May	210	218	-8	214
Jun.	228	214	14	221

Having worked through the actual sales from January to June, the model forecasts that sales from July onwards will be at the rate of 221 units per month. This method of adjusting the forecast in proportion to the size of the error is called **Single Exponential Smoothing**. Its purpose is to smooth out random variations and isolate the main pattern in the sales.

Double Exponential Smoothing takes this process a stage further by incorporating an estimate of trend into the forecast. As the name suggests, a second stage of exponential smoothing is applied to the sales forecast obtained by single exponential smoothing. The single and double smoothed values are then

combined to give a final forecast, which reflects upward or downward trends in sales.

The forward forecasts produced by Double Exponential Smoothing incorporate an estimate for the rate of change in sales and therefore each month's forecast is different. In contrast, Single Exponential Smoothing assumes no underlying trend and therefore, it gives the same forecast for each of the months ahead.

The important judgment you have to make in choosing between the single and double exponential models, is whether a trend really is present in the data.

In both models, the number of months used to create the forecast is dependent on the amount of data in the data base. If there are between 6 and 11 months available, the models use the last 6 months data. If there are 12 or more months available, the models use the last 12 months data.

The fraction of the error which is added to the old forecast at each stage of the forecasting sequence is called the smoothing constant. It can take any value between 0 and 1. If a large smoothing constant is used, the effect is to emphasise the most recent data so the model becomes very sensitive to change. Thus if sales in one month suddenly shoot up, the forecasting model will behave as if this is an established trend and will forecast that it will continue for ever.

Program description

FORECASTER first parts company from the previous program **GRAPHPLOTTER** at Line 2017 which asks the operator if he wants to forecast. If yes, he is then invited to choose between a single or double exponential forecast and flag **FO** is set to either **1 or 2**. Line 2031 checks that sufficient data (i.e. a minimum of six periods) has been entered. If there is not enough, the operator is told that he cannot have a forecast. The next stage is to enter the smoothing constant (**CO** in Lines 2048 to 2062. Note flag **DE** in Line 2048 which admits a decimal point. **CO** must be between **0 and 1** and if the entry is not within these limits, it is rejected in Line 2060 and the program returns to Line 2048 for another entry.

Data entry is now complete and the forecasting model, located in subroutine 8750, called up in Line 2064.

The principles of the model have already been described.

Lines 8752 and 8754 set **K**, the number of periods used for the forecast to either **6** or **12** depending on how much data is available. Lines 8760 to 8764 set the starting values of **P( )**, the single exponential means, **R( )**, the double exponential means and **Q( )**, the single exponential forecasts.

The forecast is calculated between Lines 8768 and 8792. Lines 8794 to 8806 align the forecast values with the data matrix **D( )** and Line 8804 stops the forecast from going negative on the grounds that negative sales are a rarity! Finally Line 8808 resets the number of periods in **D( )** to **6** more than were originally entered, so as to provide space for the forecast.

Returning now to the main program, the six month forecast is displayed in Lines 2500 to 2530. The program ends by offering a choice between restarting and exiting (Lines 3000 to 3050).

```
1000 REM FORECASTER
1001 CD$="
"
1002 FY=0:FL=0:DE=1
1004 PRINT"
":PRINT"==========
"
1006 PRINT"
ENTER NUMBER OF DATA POINTS
          (INTEGER NUMBER BETWEEN";
1007 PRINT" 2 AND 36)"
1008 PRINT"
";:D=6:GOSUB 7000
1010 IF RF=1 OR RF=2 THEN GOTO 1020
1011 IF RF=3 THEN GOTO 1016
1012 B$="INTEGER NUMBER BETWEEN 2 AND 36
":GOSUB 9000
1014 GOTO 1004
1016 GOSUB 9500
1018 GOTO 1004
1020 IF TV<36 AND TV>2 AND TV=INT(TV) TH
EN GOTO 1024
1022 PRINT"
INTEGER NUMB
ER BETWEEN 2 AND 36":GOTO 1008
1024 PRINT"
                    "
1026 NM=TV:DIM D(NM+6)
1028 PRINT"
DATA ENTRY":PRINT"==========
"
1030 PRINT"( PRESS E TO EDIT )"
1032 FOR J=1 TO NM
1034 IF J=1 THEN GOTO 1042
1036 FOR N=21-J TO 19
1038 IF N>0 THEN PRINT"
";LEFT$(CD$,N);"
                                    ";
1039 PRINT"
";LEFT$(CD$,N);"
";D(J-(20-N))
1040 NEXT N
1042 PRINT"
";LEFT$(CD$,20);"
";
1044 PRINT"ENTRY NO. ";J;"....
";
1046 IF FY>0 THEN T$=STR$(D(J)):CH=LEN(T
$):PRINT T$;:D=6:GOSUB 7004:GOTO 1049
```

```
1048 D=6:GOSUB 7000
1049 IF RF=1 THEN GOTO 1064
1050 IF RF=2 THEN GOTO 1062
1051 IF RF=3 THEN GOTO 1058
1052 IF RF=4 THEN GOTO 1072
1053 IF FY=1 THEN GOTO 1064
1054 B$="ENTER THE Y-COORDINATE OF THE P
OINT":GOSUB 9000
1056 GOTO 1042
1058 GOSUB 9500
1060 GOTO 1042
1062 PRINT"■■■■■■■■■■■■■■■■■■■■NUMBER GREAT
ER THAN SIX CHARS !":GOTO 1042
1064 PRINT"■■■■■■■■■■■■■■■■■■■■
                    "
1066 D(J)=TV
1068 NEXT J
1070 GOTO 1076
1072 GOSUB 6000
1074 GOTO 1034
1076 IF FY=1 THEN FY=0
1078 PRINT"■"
1080 PRINT"DO YOU WANT TO CORRECT THE EN
TRIES ?";
1082 GOSUB 8500
1083 IF RF=1 OR RF=2 THEN GOTO 1094
1084 IF RF=3 THEN GOTO 1090
1086 B$="IF Y THEN EDIT.IF N THEN CARRY
ON":GOSUB 9000
1088 GOTO 1078
1090 GOSUB 9500
1092 IF TV=1 THEN GOTO 2000
1094 IF TV=2 THEN GOTO 2016
2000 REM CHECKING ROUTINE
2002 PRINT"■"
2004 PRINT"CHECK ON ENTRY":PRINT"=======
=======
2006 PRINT"IF O.K. THEN RETURN"
2008 PRINT"IF WRONG THEN RE-ENTER"
2010 PRINT"PRESS E TO EDIT"
2014 FY=1:GOTO 1032
```

```
2016 REM TO FORECAST OR NOT
2017 PRINT"■DO YOU WANT A FORWARD FORECA
ST ?"
2018 IF FL=1 THEN PRINT"YOU CANNOT DO A
FORECAST WITH LESS THAN 6 POINTS"
2019 GOSUB 8500
2020 IF RF=1 THEN GOTO 2029
2021 IF RF=3 THEN GOTO 2026
2024 GOTO 2018
2026 GOSUB 9500
2028 GOTO 2018
2029 IF TV=1 THEN GOTO 2031
2030 IF TV=2 THEN GOTO 2500
2031 IF NM<6 THEN FL=1:GOTO 2018
2032 PRINT"■DO YOU WANT":PRINT"1) A SING
LE EXPONENTIAL FORECAST"
2033 PRINT"2) A DOUBLE EXPONENTIAL FOREC
AST■";LEFT$(CD$,16);"ENTER 1 OR 2 ";
2034 D=1:GOSUB 7000:IF RF=1 THEN GOTO 20
44
2035 IF RF=3 THEN GOTO 2040
2036 B$="SEE TEXT FOR AN EXPLANATION OF
THE DIFFERENCE":GOSUB 9000
2038 GOTO 2032
2040 GOSUB 9500
2042 GOTO 2032
2044 FO=TV
2046 IF TV<1 OR TV>2 THEN GOTO 2032
2047 PRINT"■"
2048 PRINT"■ENTER THE SMOOTHING CONSTANT
 ( 0 -> 1 )■■■■■■■■";:D=4:DE=1:GOSUB 7000
2049 IF RF=1 THEN GOTO 2060
2050 IF RF=2 THEN GOTO 2062
2051 IF RF=3 THEN GOTO 2056
2052 B$="THE CONSTANT CAN BE UP TO 4 CHA
RACTERS  LONG":GOSUB 9000
2054 GOTO 2048
2056 GOSUB 9500
2058 GOTO 2048
2060 IF TV>0 AND TV<1 THEN DE=0:CO=TV:GO
TO 2064
```

```
2062 PRINT"■■■■■■■■■■■■■■■■■■■■CONSTANT IS
OUTSIDE PERMITTED RANGE":GOTO 2048
2064 PRINT"■":GOSUB 8750
2500 REM DISPLAY FORECAST SIX POINTS
2505 PRINT"■FORECAST AHEAD FOR NEXT SIX
DATA POINTS"
2510 FOR L=NM-5 TO NM
2515 PRINT"POINT ";L;".  ";D(L)
2520 PRINT
2530 NEXT L
3000 PRINT"■";LEFT$(CD$,18);"PRESS 1 TO
RESTART OR 2 TO EXIT :"
3010 D=1:DE=0:PRINT" ■";:GOSUB 7000
3020 IF RF=3 THEN GOSUB 9500:GOTO 3000
3030 IF RF<>1 THEN GOTO 3000
3040 IF TV=1 THEN RUN
3050 PRINT"■":END
6000 REM EDIT ROUTINE
6002 L=J:IF L>19 THEN L=19
6006 PRINT"■";LEFT$(CD$,20-L);"■■■■■■■■■■
■■■■■■■■■■■■■■CORRECTION"
6008 PRINT"■";LEFT$(CD$,21-L);"■■■■■■■■■■
■■■■■■■■■■■■■■■■■■*■■";:D=6:GOSUB 7000
6009 IF RF=1 THEN GOTO 6016
6010 IF RF=2 THEN GOTO 6022
6011 IF RF=3 THEN GOTO 9300
6012 PRINT"■";LEFT$(CD$,21-L);"■■■■■■■■■■
■■■■■■■■■■■■■■■■■■ ";
6013 L=L-1:IF L<2 THEN GOTO 6024
6014 GOTO 6008
6016 PRINT"■■■■■■■■■■■■■■■■■■■■";
6018 D(J-L+1)=TV:GOTO 6013
6020 REM
6022 PRINT"■■■■■■■■■■■■■■■■■■■■NUMBER GREATE
R THAN SIX CHARS":GOTO 6008
6024 PRINT"■";LEFT$(CD$,20-L);"■";:RETUR
N
7000 REM NUMBER INPUT
7002 TV=0:T$=" ":CH=0
7004 GET A$:IF A$<>"" THEN GOTO 7004
```

```
7005 GET A$:IF A$=""THEN GOTO 7005
7006 IF ASC(A$)=13 AND CH=0 THEN RF=0:GO
TO 7040
7010 IF ASC(A$)=20 THEN GOTO 7030
7012 IF ASC(A$)=13 THEN RF=1:GOTO 7040
7014 IF A$="@" AND CH=0 THEN RF=3:GOTO 7
042
7016 IF CH=D THEN RF=2:GOTO 7042
7018 IF A$="E" THEN RF=4:GOTO 7042
7020 IF A$>="0" AND A$<="9" THEN GOTO 70
26
7022 IF DE=1 AND A$="." THEN GOTO 7026
7024 GOTO 7004
7026 T$=T$+A$:PRINT A$;:CH=CH+1
7028 GOTO 7004
7030 IF CH=0 THEN GOTO 7004
7032 CH=CH-1:PRINT"■■ ■■";
7034 IF CH=0 THEN T$="":GOTO 7004
7036 T$=LEFT$(T$,LEN(T$)-1)
7038 GOTO 7004
7040 TV=VAL(T$)
7042 D=0:RETURN
8000 REM SPACE BAR INPUT
8002 TV=0:T$=" ":CH=0
8004 GET A$:IF A$=" " THEN GOTO 8004
8005 GET A$:IF A$<>" " THEN GOTO 8005
8034 RF=1:TV=1:RETURN
8500 REM YES OR NO INPUT
8005 GET A$:IF A$<>" " THEN GOTO 8005
8034 RF=1:TV=1:RETURN
8500 REM YES OR NO INPUT
8502 RF=0:CH=0
8504 GET A$:IF A$<>""THEN GOTO 8504
8505 GET A$:IF A$="" THEN GOTO 8505
8506 IF A$="@" THEN RF=3:GOTO 8514
8508 IF A$="Y" THEN RF=1:TV=1:GOTO 8514
8510 IF A$="N" THEN RF=1:TV=2:GOTO 8514
8512 GOTO 8504
8514 PRINT A$:RETURN
8750 REM FORECASTING MODEL
```

```
8752 IF NM<12 THEN K=6
8754 IF NM>=12 THEN K=12
8756 DIM P(12),Q(19)
8758 DIM R(12),S(19)
8760 P(1)=D(NM-K)
8762 R(1)=P(1)
8764 Q(2)=P(1)
8766 REM SMOOTHING
8768 FOR H=1 TO K-1
8770 A=NM-K+H
8772 P(H+1)=P(H)+CO*(D(A)-P(H))
8774 Q(H+2)=P(H+1)
8776 R(H+1)=R(H)+CO*(P(H+1)-R(H))
8778 A=2*P(H+1)-R(H+1)
8780 B=CO*(P(H+1)-R(H+1))/(1-CO)
8782 S(H+2)=A+B
8784 NEXT H
8786 FOR H=1 TO 6
8788 Q(K+H+1)=P(K)
8790 S(K+H+1)=A+B*(H+2)
8792 NEXT H
8794 FOR H=1 TO 6
8796 IF FO=1 THEN GOTO 8798
8797 IF FO=2 THEN GOTO 8802
8798 D(NM+H)=Q(K+H+1)
8800 GOTO 8804
8802 D(NM+H)=S(K+H+1)
8804 IF D(NM+H)<0 THEN D(NM+H)=0
8806 NEXT H
8808 NM=NM+6
8810 RETURN
9000 REM HELP ROUTINE
9002 PRINT"SQQQQQQQQQQQQQ
                                 ";
9003 PRINT"
          ";
9004 PRINT"
          ";
```

```
9005 PRINT"■■■■■■■■■■■■■■■■■■■■";B$:PRINT"PR
ESS SPACE TO CONTINUE":GOSUB 8000
9006 IF RF=1 THEN GOTO 9014
9008 GOTO 9002
9010 GOSUB 9502
9012 GOTO 9002
9014 PRINT"■■■■■■■■■■■■■■■■■■■■
                                ";
9015 PRINT"
          ";
9016 PRINT"
          "
9018 RETURN
9500 REM ESCAPE ROUTINE
9502 PRINT"■■■■■■■■■■■■■■■■■■■■DO YOU WANT T
O EXIT (Y/N)":GOSUB 8500
9503 IF RF=1 THEN GOTO 9506
9504 IF RF=2 OR RF=3 THEN GOTO 9502
9506 IF TV=2 THEN GOTO 9510
9508 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■THE END■■
■■■■■■■■==========":END
9510 PRINT"■■■■■■■■■■■■■■■■■■■■
                                ":RETURN
```

7

CONTACTS
Customer Records

Introduction:

As the name suggests, a database program converts a computer into an information store. Once in the store, data can be retrieved for examination and there are provisions for entering or deleting data as requirements change.

Thus far, a computer database is merely an electronic equivalent of a filing cabinet. As such it may well only offer marginal benefits as compared with tried and tested manual systems. What makes a computer database different and such an enormous improvement over manual systems is that the process of retrieving the information can be programmed. That is, the computer can be instructed to work its way through the database sorting the data in whatever way the programmer chooses. This ability to handle and analyse data is only limited by the way the data fed into the computer was coded in the first place.

The need to sort information is central to virtually all administrative and clerical tasks. Wherever you look people are telling other people to:

– List all the people who have not paid their motor car tax.

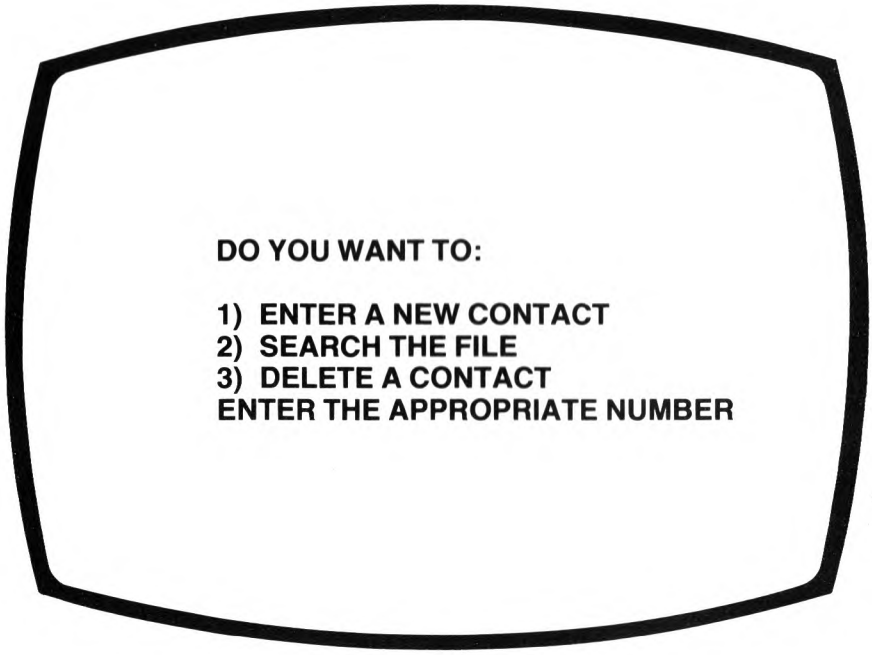
– List all the debtors who are more than 3 months overdue.

– List all the customers who have not ordered anything for 6 months.

and so on.

With manual systems, the answers are found by sorting decks of cards or worse still, going through lists of entries in a ledger. With a computer database the task is done electronically in a fraction of the time.

Database programs fall into two main categories – programs aimed at a particular application and general purpose systems. The most common applications of database programs are in accountancy and bookkeeping. General purpose systems are written to be as flexible as possible and they normally contain their own system (in effect a high level language) for enabling the operator to specify how the data are to be manipulated, how files are to be organsied and how output is to be formatted.



CONTACTS is a simple database program specifically intended to store information on a salesman's contacts. It stores single entries comprising company name, contact name and telephone number. Entries can be loaded into the database in any order and new entries can be added as each new contact is made. Data are sorted using the very common technique of "keyword" searching. This is a rather clever way of enabling the operator to dictate the scope of the search he wants carried out. The idea is that the computer searches for all the entries which begin with the keyword. The scope of the search is determined by the length of the keyword. If it comprises just one letter, it will pick out many more entries than if it is 10 letters long.

To find the name(s) of the contact(s) at a particular firm, you enter an appropriate keyword. **CONTACTS** then searches the database and lists out the entries which it finds. Keyword searching really comes into its own if there are entries which have part of their name in common. If for example there are a number of branches of the same firm [say Jones Ltd. (Leeds), Jones Ltd. (Wakefield) and Jones Ltd. (Barnsley)] all Jones branches will be listed if the keyword is "Jones" whilst "Jones Ltd. (L" will pick out just the Leeds branch.

Program description

The program is made up of four main sections:

– The main menu.

– Data entry, where the contact information is entered.

– Search, where the entries beginning with the keyword are identified and listed.

– Data deletion, where the entries which are no longer wanted are erased.

Since there are only a few keyboard entries, the subroutines for data entry have been simplified.

The main menu:

The program starts at Line 102 where **F$ ( )** – the array in which the data are to be stored, is given the dimension **B**. (**B** is set to 200 but you can of course change it if you wish). Lines 106 and 108 are the program title.

Data are permanently stored on a microdisc and are loaded into the program at the start. However, when the program is being used for the first time, this file has yet to be created and if an attempt were made to load a non-existent file, the program would 'crash'. Lines 112 to 118 get over this problem, by asking whether a file exists. If it does, it is loaded (subroutine 5500); if not the loading routine is by-passed.

Lines 126 to 139 display the menu options and record the choice that is made. Note that the value of **T$** can only be **1, 2** or **3**. If it is anything else, the program goes back to the start through Line 140. The flag **FM** is set to **1** if option 3 (**DELETE**) is chosen.

Data entry:

One complete entry comprises three pieces of information. The

firm's name, the name of the contact and his telephone number. These three separate pieces of information are combined and saved as one element of the array **F$( )**.

The program for entering data begins at Line 1000. Lines 1006 to 1018 collect the data on one contact in the form of three strings **N$, C$ and Q$**. Next, the three strings are combined into a single string, **R$**, with each piece of information separated by a dividing string like this:

"CONTACT . . . . "

Variable **D** limits the length of the data entry strings (**N$** etc). The first string is limited to 20 characters and the last two to 10. The dividing strings contain 25 characters so the maximum length of one whole entry is 65 characters – that is rather less than two lines on the 40 column screen.

NO in Line 1020 is a counter which registers the number of entries by advancing by one every time an entry is made. Provided there is room in the array, **R$** becomes **F$(NO)**, that is, the **NO**th element of the array **F$( )**, in Line 1026. When **NO** is equal to the dimensions of **F$( )**, the file is full and Line 1022 prevents any more entries being accepted. To do this, the program assumes that data entry is finished and proceeds to by-pass Lines 1026 to 1030.

When an entry has been completed, there is a choice between making another entry or finishing with data entry altogether and returning to the main menu. The first branch (**RF=1**), Line 1032, returns to the start of the data entry routine at Line 1002. The second branch (**RF=2**) first saves the complete record **F$( )** on microdisc in subroutine 5000 before returning to the main menu.

Search:

The search routine begins by asking for a "**keyword**" (Lines 2004 to 2008), finding its length, **LE**, (Line 2008) and then scanning through **F$** to find entries whose first **LE** characters are identical to **S$**. The search takes place in a loop which runs from **1 to NO**,

between Lines 2012 and 2016. Each element of **F$(N)** is extracted in turn (Line 2014), its first **LE** characters are isolated and the resultant string compared with **S$**. If the first **LE** characters of **F$(N)** are the same as the keyword, the program branches to subroutine 6000 which prints **F$(N)** in full (Line 6008). If they are not the same, the program 'drops through' Line 2014 to 2016 and advances the loop to pick up the next entry in **F$(N)**.

There are a number of special conditions which have to be catered for. It is quite possible to find more entries under a particular keyword than the screen can cope with. If this were to happen, the screen would scroll so causing lines of output to be lost from the top of the screen. To avoid this, scrolling is brought under operator control in Lines 6000 to 6008 by restricting the number of entries which can be printed to **5**. (This is the number of entries that can be conveniently fitted onto the screen if they are all 2 lines in length. If **L** becomes greater than **5**, the program is stopped until the operator presses the space bar at which point the counter **L** is set back to **0** and the screen is cleared with a **clear screen** or CLS command (Line 6007). The flag **FL** in Line 6008 caters for the eventuality that there are no entries which match the keyword in the whole of **F$(N)**. **FL** is set to **0** at the start of the search routine in Line 2002. When an entry has been found, **FL** is set to **1** in Line 6008. If at the end of the search, **FL** is still **0** the "**no-one there**" message in Line 2018 is triggered.

The flag **FM** (Lines 2015 and 2026) can be ignored for the moment. It is part of the **DELETE** routine which is described later. **FM** has the value **0** during the search routine.

Searching ends by asking "**do you want another search**", Line 2020. If the answer is **Y** (yes), the program returns to the start of the search routine at Line 2000. If **N** (no), it returns to the main menu at Line 122.

Delete:

Deletion is a branch of the search routine. Option (3) in the main menu leads onto Line 142 where flag **FM** is set to **1**. Thereafter, the operator specifies the name to be deleted and the search is done by the same program as is used for Option (2). When the first entry

is found (Line 2015), **FM** flags the program into subroutine 3000.

Lines 3002 and 3004 ask whether this particular entry is to be deleted or not. If it is to be deleted, the particular value of **F$( )** is set to "" (the nul value) in Line 3006. If the entry is not to be deleted, the program returns to Line 2016 and initiates another loop in the search routine. If a deletion has taken place, the remaining entries in **F$( )** have to be moved up to eliminate the space left by the deletion. This is done in Lines 3008 to 3016. The value of **N** gives the position of the **F$( )** entry which has just been deleted. Line 3010 assigns to **F$(N)** the value of **F$(N+1)** (that is the entry immediately following the one which has just been deleted). This process is repeated for all the elements between **N** and the end of the entries at **NO** using the for/next loop in Lines 3008 to 3012. When the 'reshuffle' has been finished, there is one entry at the very end of the array which should not be there. It is deleted in Line 3014. Finally, since an entry has been deleted, **NO** (the number of entries) is reduced by one (Line 3016).

The program now returns to the search routine and goes around the search loop until all the entries have been found. **DELETE** again parts company from the search program at Line 2026 when the operator signifies that no more searches are needed and the program branches into subroutine 5000 where the microfloppy record is rewritten. Finally the operator is returned to the main menu via Line 2028.

```
10 REM CONTACTS DATA BASE PROGRAM
12 NO=0
14 REM C$=CONTACT NAME STRING
16 REM D= STRING LENGTH
20 REM F$=FILE DATA
22 REM FM FLAGS DELETE INSTRUCTION
24 REM L= PAGE COUNTER
26 REM LE=LENGTH OF SEARCH STRING
28 REM M AND N ARE COUNTS IN FOR/NEXT LO
OPS
30 REM N$=NAME STRING
32 REM NO=NUMBER OF FILE ENTRIES
34 REM R$=RECORD
36 REM RF FLAGS EXIT FROM DATA ENTRY SUB
ROUTINES
38 REM S$=SEARCH STRING
40 REM Q$=TELEPHONE NUMBER STRING
42 REM T$=TRANSFER STRING FROM DATA ENTR
Y SUBROUTINES
100 REM CONTACTS FILE
102 B=200:DIM F$(B)
106 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■CONTACTS"
108 PRINT"■■■■■■■■■■==========="
110 REM LOAD CONTACTS
112 PRINT"■■■■■■■■■■■■■■■■DO YOU ALREADY H
AVE A CONTACTS FILE ?"
114 PRINT"■■■■Y OR N";
116 GOSUB 8500
118 IF RF=1 THEN GOSUB 5500
122 REM MAIN MENU
124 PRINT"■":FM=0
126 PRINT"■■DO YOU WANT TO :"
128 PRINT"■■1)  ENTER A NEW CONTACT"
130 PRINT"■■2)  SEARCH THE FILE"
132 PRINT"■■3)  DELETE A CONTRACT"
134 PRINT"■■■ENTER THE APPROPRIATE NUMBE
R :   ■■■";
135 D=1:GOSUB 7000
136 IF T$<"1" OR T$>"3" THEN GOTO 135
137 IF T$="1" THEN GOTO 1002
```

```
138 IF T$="2" THEN GOTO 2000
139 IF T$="3" THEN GOTO 142
140 GOTO 134
142 FM=1:GOTO 2000
1000 REM PRINT CONTACTS
1002 REM ENTER CONTACT INFORMATION
1004 PRINT"■ENTER CONTACT DETAILS":PRINT
"====================="
1006 PRINT"■■■■NAME.......";:D=20:GOSUB
7000:PRINT
1008 N$=T$
1010 PRINT"■■CONTACT....";:D=10:GOSUB 70
00:PRINT
1012 C$=T$
1014 PRINT"■■TEL. NO....";:D=10:GOSUB 70
00:PRINT
1016 Q$=T$
1018 R$=N$+"  CONTACT..."+C$+" TEL. NO..
.."+Q$
1020 NO=NO+1
1022 IF NO<B+1 THEN GOTO 1026
1024 PRINT"■■■■■■■■■■■■FILE FULL. YOU CANN
OT ENTER ANOTHER      CONTACT. ";
1025 PRINT"ENTER N TO RETURN TO MENU":GO
TO 1031
1026 F$(NO)=R$
1028 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■DO YOU W
ANT TO ENTER ANOTHER CONTACT ?"
1030 PRINT"■■Y OR N";
1031 GOSUB 8500
1032 IF RF=1 THEN GOTO 1002
1034 GOSUB 5000
1036 GOTO 122
2000 REM SEARCH ON NAME
2002 FL=0:L=0
2004 PRINT"■SEARCH■■■■■■■=======
2006 PRINT"■ENTER NAME    ";:D=20:GOSUB 7
000
2008 S$=T$:LE=LEN(S$)
2009 PRINT
```

```
2010 PRINT"■■CONTACTS ON FILE ARE :"
2012 FOR N=1 TO NO
2014 G$=F$(N):IF LE>LEN(G$) THEN GOTO 20
16
2015 IF LEFT$(G$,LE)=S$ THEN GOSUB 6000:
IF FM=1 THEN GOSUB 3000
2016 NEXT N
2018 PRINT"END OF SEARCH";:IF FL=0 THEN
PRINT".....NO ONE THERE"
2020 PRINT:PRINT"DO YOU WANT TO DO ANOTH
ER SEARCH ?       (Y OR N)"
2022 GOSUB 8500
2023 IF RF=1 THEN GOTO 2000
2026 IF FM=1 THEN GOSUB 5000
2028 PRINT"■":GOTO 122
3000 REM DELETE CONTACTS
3002 PRINT"DELETE ? (Y OR N)":GOSUB 8500
3004 IF RF=2 THEN GOTO 3018
3008 FOR M=N TO NO
3010 F$(M)=F$(M+1)
3012 NEXT M
3014 F$(M)=""
3016 NO=NO-1
3018 RETURN
5000 REM SAVE CONTACTS FILE
5010 OPEN 1,1,1,"DATAFILE"
5020 PRINT#1,NO
5022 RETURN
5030 FOR L=1 TO NO
5040 PRINT#1,F$(L)
5050 NEXT L
5060 CLOSE 1
5500 REM LOAD CONTACTS FILE
5510 OPEN 1,1,0,"DATAFILE"
5516 RETURN
5520 INPUT#1,NO
5530 FOR L=1 TO NO
5540 INPUT#1,F$(L)
5550 NEXT L
5560 CLOSE 1
```

```
6000 REM PRINT CONTACT LIST
6002 IF L<5 THEN GOTO 6008
6004 PRINT"PRESS SPACE BAR TO CONTINUE"
6006 GET A$:IF A$<>" " THEN GOTO 6006
6007 L=0:PRINT"■"
6008 PRINT F$(N):FL=1
6010 L=L+1:RETURN
7000 REM DATA ENTRY ACCEPTS ALL KEYBOARD
 CHARACTERS
7002 T$="":CH=0
7004 GET A$:IF A$<>"" THEN GOTO 7004
7005 GET A$:IF A$="" THEN GOTO 7005
7006 IF CH>0 THEN GOTO 7032
7008 IF ASC(A$)=13 AND CH>0 THEN GOTO 70
32
7010 IF ASC(A$)=20 THEN GOTO 7018
7012 IF A$>" " AND A$<"Z" THEN GOTO 7028
7016 GOTO 7004
7018 IF CH=0 THEN GOTO 7004
7020 CH=CH-1:PRINT"■■ ■■";
7022 IF CH=0 THEN T$="":GOTO 7004
7024 T$=LEFT$(T$,LEN(T$)-1)
7026 GOTO 7004
7028 T$=T$+A$:PRINT A$;:CH=CH+1
7030 GOTO 7004
7032 RETURN
8500 REM Y OR N ENTRY
8502 CH=0:RF=0
8504 GET A$:IF A$<>"" THEN GOTO 8504
8505 GET A$:IF A$="" THEN GOTO 8505
8506 IF CH>1 THEN GOTO 8514
8508 IF A$="Y" THEN RF=1:GOTO 8514
8510 IF A$="N" THEN RF=2:GOTO 8514
8512 GOTO 8504
8514 PRINT A$:RETURN
```

8

WHO'S Selling What?

Introduction:

CONTACTS (Chapter 7) is a very simple database program. Each record is held as a single string and when the records are sorted, they remain in the same form as they were entered. That is, the program sorts on just one "**Field**"; it only searches for the contact name and not the company name or telephone number.

For more general applications, a database program must be able to reorganise raw data and display it in the different forms needed for analysis. Thus a database program has three main functions: to set up and maintain ongoing files of data (the database), to reorganise the data and to provide a means of displaying output in a form specified by the operator.

In this program, data are recorded as each sale takes place. When an analysis is required, **WHO'S** produces tables showing different arrangements of the original data. The data are rearranged using the '**chain array**' technique described in Chapter 3.

The database comprises a number of sales data files each of which contains a complete record of a month's transactions. Part of this record comprises data which are unique to the transaction itself (the sales amount) but most of it also appears on other transactions, (i.e. the date, salesman name etc). Instead of duplicating these common data throughout the files, they are stored in full only once, in '**directories**' which are referenced by an entry in the sales data file. Thus instead of entering the salesman's name each time a transaction is recorded, all that is entered is a numerical code which refers to an entry in the salesman directory.

Each time a transaction is entered, the following data are stored on the sales data file:

– The date the sale was made.

– The salesman's number (cross referenced to a name file).

– The customer number (cross referenced to a name file).

– The product code (cross referenced to a description).

– The amount of the sale.

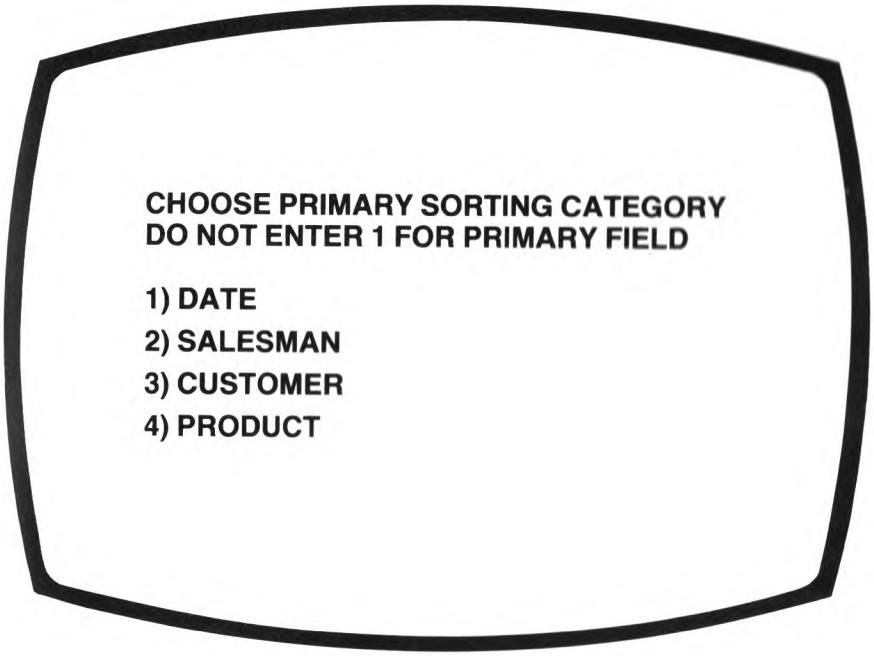
Three directory files contain the following:

– Salesman names

– Customer names.

– Product names (or descriptions).

WHO'S



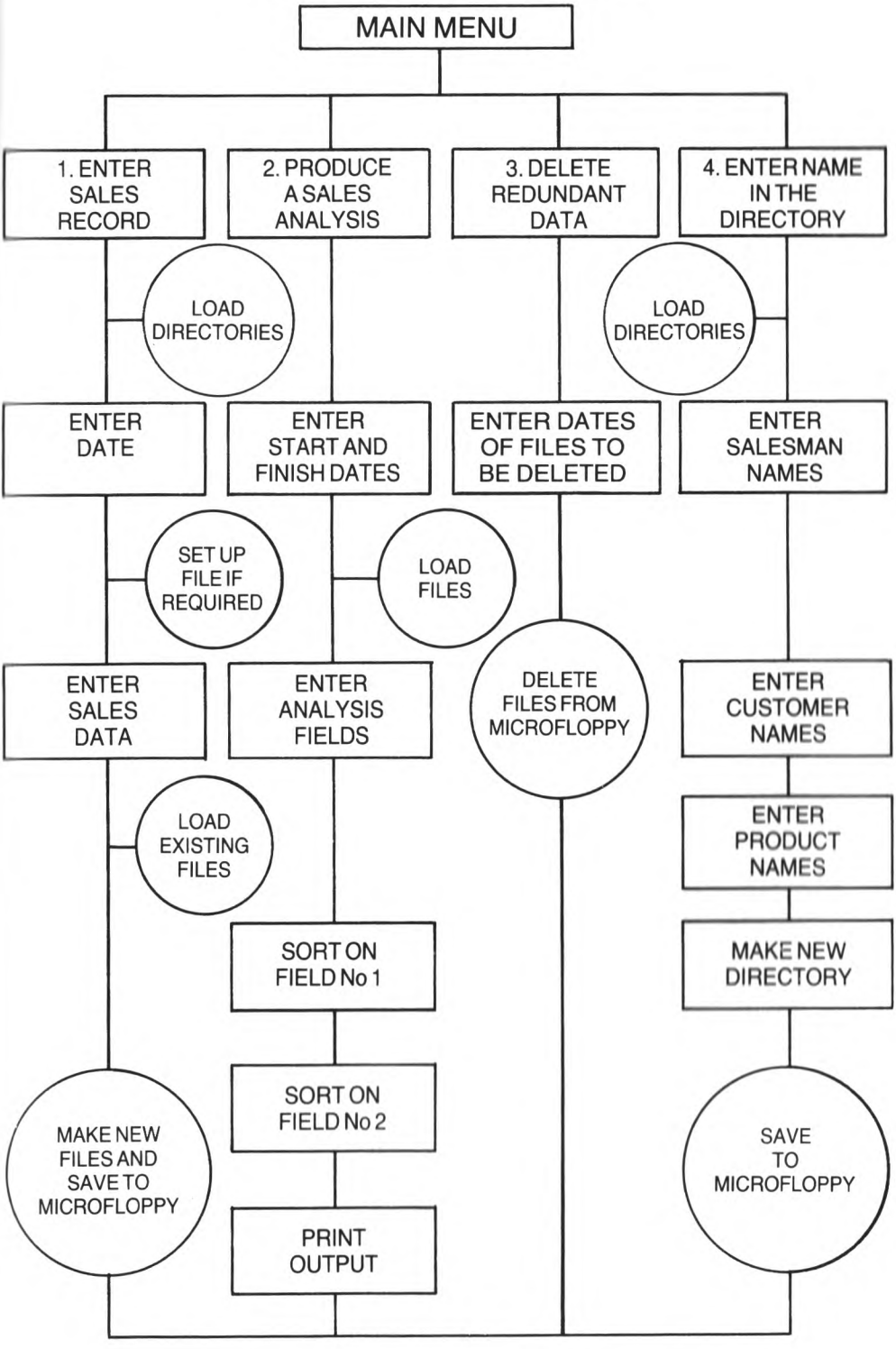
To analyse the database, the operator specifies the period he wishes to examine by entering the year and month of the start and finish dates. The program then loads the contents of the appropriate sales data files into memory. Next, the operator chooses the fields on which the data are to be analysed. For example, he might choose to analyse by customer and then by date, in which case the data would first be divided into sales by customer (all sales to customer 1, then all sales to customer 2 and so on) and secondly ranked in date order within each customer category. (All customer 1 transactions listed in date order followed by customer 2 transactions in date order and so on.)

The quantity of data which can be analysed at any one time is of course determined by the amount of computer memory available. Since a number of different files have to be fitted into memory, the space assigned to each file must be maintained in some sort of balance. Clearly, it is not desirable to have room for hundreds of customers and only a few transactions. The best arrangement depends on the characteristics of the particular business being set up on the database. File size is determined by the values of the constants and the dimensions of the arrays and matrices set in the first few lines of the program.

The issue of "**idiot proofing**" is of major importance in this type of program. Should the user be prevented from putting the wrong cartridge in the disc drive? What happens if he calls for a data file which does not exist? All eventualities can be catered for but they considerably lengthen and complicate the program. **WHO's** takes a compromise position by protecting against likely operator errors but it stops short of preventing deliberate attempts to 'fool' the machine.

The structure is outlined on the following flow chart:

WHO'S SIMPLIFIED BLOCK DIAGRAM



Program description:

WHO'S begins with a menu offering a choice between:

1) ENTER A SALES RECORD
2) PRODUCE AN ANALYSIS OF SALES
3) DELETE REDUNDANT DATA
4) ENTER NAMES IN DIRECTORIES
5) SET UP NEW DATABASE
6) QUIT

Set up:

Option (5) is only used when an entirely new database is to be set up. Its purpose is to 'open' the directory files into which data will be loaded at a later stage. This avoids a situation where the computer is unable to "find" a file when it is asked to save the first directory entries. **SET UP** creates dummy files each of which contains a single entry (the number -999999) as an indication that the file is newly created.

Option (5) calls subroutine 6000 which, after warning the operator that any files he may have on the disc will be erased, creates three files **SADIR** (salesman directory), **CUDIR** (customer directory) and **PRDIR** product directory). Each file is given a single -999999 entry.

Making the directories

Option (4) loads data into the three directories. Initially, the file contains no data (just a –999999). However, once the database is in use, the directory will contain a list of names which may need to be amended and added to as time goes on. The maximum number of entries which can be held in each directory is determined by the value assigned to variables **NS**, **NC** and **NP** in Lines 102 to 106.

Option (4) calls subroutine 6500 the first stage of which sets up a loop which opens and then reads each of the three directory files, **SADIR**, **CUDIR** and **PRDIR**. An 'empty file is recognised if the first entry is -999999 in which case it is immediately closed, Line 6514. If the file already contains records, the first entry is the number of records on file (**NR**). These are read into any array **B$( )** in Lines 6516 to 6520.

What follows is very similar to the data entry routine used in **GRAPHPLOTTER** (Chapter 5). The **N** loop which runs from Line 6516 to 6520 counts the number of records to be entered into array **B$ ( )** so that at the end of the loop, **B$( )** contains the **NR** records which have been loaded from the existing file. At this point flag **FY** is set to **1** (Line 6524). These "old' records will form the first part of the new directory so, before entering new data they must be displayed and an opportunity provided to amend them. The contents of **B$( )** are displayed by feeding each element of **B$( )** into the input subroutine 8550 as if they were keyboard entries. This is done by setting **T$** and **CH** to the value of the appropriate element of **B$( )** in Line 6542 and then entering the input subroutine at Line 8554. Subroutine 8550 then takes over and treats the entry as if it were a partially completed keyboard entry and waits to receive another value of **A$**. To amend an entry, the operator back spaces and enters new data. A **RETURN** entry causes an **RF=1** exit from subroutine 8550 so transferring the contents of **T$** to **B$( )** and placing the original or amended data in the new directory.

When all the old entries (**NR** of them) have been displayed, **J<NR+1** (Line 6542), ceases to be effective and the next entry falls through to Line 6544 which cancels editing (**FY=0**) and increases the value of **NR** to **NT** so allowing the edit routine to be used on the new data about to be entered. Note that **NT** was set to the maximum capacity of the directory in Line 6504.

If, at any stage in the data entry routine, the operator wishes to edit his entry, he presses the * key. This causes an **RF=4** exit from subroutine 8550 which in turn enters subroutine 6750. Subroutine 6750 works through the entries so far made (of which there are **L**), feeding each one into the data entry routine (subroutine 8550).

Entries can be (1) accepted (press return for an **RF=Ø** exit, which triggers another circuit of the loop in Line 6764) or (2) amended, in which case the return is through **RF=1** and the appropriate element of **B$( )** is assigned the new value or finally (3) they can be incorrect, in which case **RF=2** leads into Line 6772 for a prompt before returning to Line 6758. When editing is complete, the program exits from subroutine 675Ø, returns to Line 6572 and thence to Line 6532 to pick up the next entry in the directory.

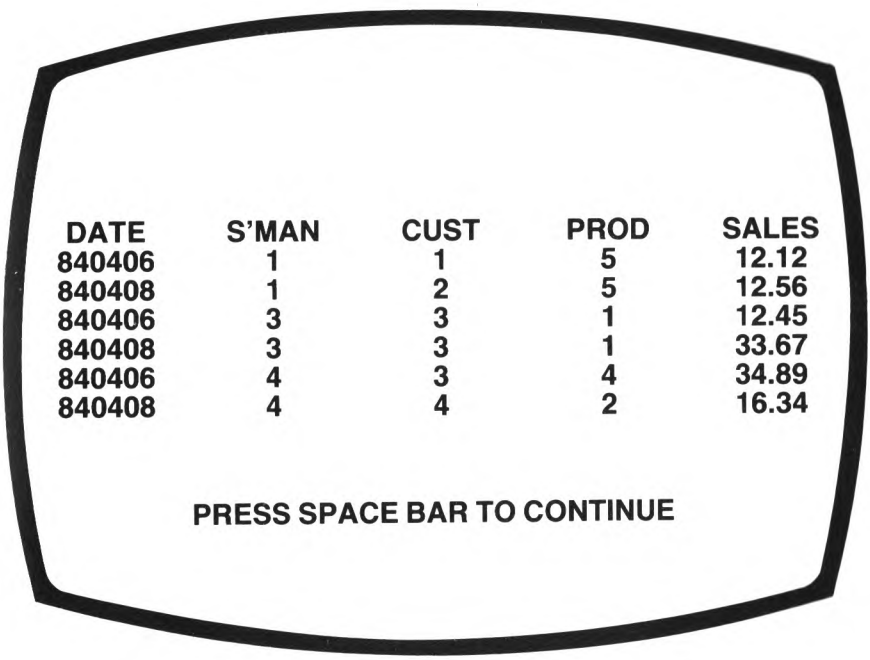
The operator signals that he has finished data entry by pressing the "." key. This causes an **RF=5** exit from subroutine 855Ø (Line 6551) and he is once again asked if he wishes to edit. If he does, the program repeats the whole of the directory entry routine by returning to Line 653Ø after setting **FY** to **1**. The effect of this is to display the new values **B$( )** so that the operator can add to or amend his entry. When he indicates that data entry is finally complete, (i.e. he does not want to correct Line 6578), he triggers an exit to Line 66Ø6 where the first directory file **SADIR** is created.

When the file has been made, new values for **D$** and **J$** are entered (Line 6622) and the whole process repeated for **CUDIR** and **PRDIR**, the remaining directory files. When all three directory files have been saved, the operator is returned to the main menu in Line 6626.

Entering sales data:

Sales data can be entered on an on going basis as and when they become available. As a transaction is entered, each item of information is routed to the appropriate column of a five column matrix **R$(,)**. The first column (column 1) contains the date, the second the salesman's number, the third the customer number, the fourth the product number and the fifth the sales amount. Sales records are accumulated in a file, the name of which comprises the year and month the data were entered. (For example, FILE 83Ø9 covers year 83, month 9).

WHO'S



The data entry routine starts in Line 15ØØ by loading the contents of the three directories (salesman, customer and product) into arrays **S$( )**, **C$( )** and **P$( )** under the control of flag **FL**. The number of entries in each directory is held in **N(1)** to **N(3)**.

The next stage is to ask the operator whether this is the first entry in the month. If the answer is **Y** (yes), a flag **(FL)** is set to **1** (Line 1552). This flag triggers the creation of a dummy (-999999) file once the year and month of the date has been entered via Line 1582 which calls subroutine 89ØØ.

The date entry routine follows the familiar pattern described in earlier programs. In **WHO's**, the part of the routine which handles the entry of the year and month is put into subroutine 87ØØ. Note that where necessary, the month and day numbers are artificially made up to two characters. (Lines 1578 and 874Ø).

Once the date has been entered, the transaction details are entered in an **M** loop starting at Line 1590. The number of transactions which can be entered in one "go" being limited by the value of **A**. (Which can be set to any value up to the dimension of **R$(,)**, itself set in Line 114).

Line 1586 to Line 1688 follows the usual pattern. Flag **FY**, set to zero at the start, becomes **1**, if the edit mode is selected later on. Flag **DE** is used to control the admission of the decimal point in subroutine 7000. The Salesman, Customer and Product codes are handled by asking the operator to enter the appropriate reference number (e.g. the salesman's number). When the number is entered, the complete directory entry (i.e. the salesman's name), held in **S$( )**, is displayed alongside the reference number. This first takes place in Line 1620. Note the check to see if the number lies within the directory's range which first occurs in Line 1618. There is an opportunity to edit at the end of data entry (Line 1724) when the operator is asked whether he wishes to correct the entries. If he does, flag **FY** is set to **1** and the upper limit on the **M** loop (variable **A**) is set to **NE**, the number of entries, and the program then returns to the start of the routine at Line 1590. Finally, the **NE** entries in **R$(,)** are saved on a file whose name is the year and month entered at the start of data entry (Line 2000).

A new file is now made by first loading all the data which are already on file into **R$(,)** just 'behind' the data which have just been entered for the first time. To do this the counter **N** is set to start at **NE+1**, the first vacant element, and to finish at **NE+NR**, which is now the total number of records. The final stage is to re-make the file by saving all the elements of **RE$(,)** from **1** to **NT** on the microfloppy disc. (Lines 2030 to 2048)

Sales analysis:

Option (2) calls subroutine 3000, the purpose of which is to analyse the sales records and produce an operator selected, tabular listing of the transactions which took place between two specified dates.

The first stage is to enter the two dates, the start and finish, which determine which of the appropriate monthly files should be loaded into memory. This is done between Lines 3000 and 3054. The start

and finish dates (year and then month) are entered using subroutine 8700. The start date is converted to equivalent numerical variables **T** and **D** which count the date forward until it becomes equal to the finish date, Lines 3048 to 3054. Each file whose date lies between the start and finish is opened and its contents loaded into **R$(,)** using **NT** and **NE** to position the data.

Since **R$(,)** will contain data from more than one monthly file, Line 3042 adds the year and month to the transaction day held in **R$(N,1)** as each file is read.

With the data in place, the next stage is to analyse it. To explain this requires a further look at how chains are used.

Chain sorting:

Suppose that the operator has chosen to sort the data into the product categories contained in column 4 of **R$(,).** To do this, each row of the matrix must be repositioned so that column 4 is in a descending numerical order. This is illustrated in the diagram below where the matrix:

ROW	DATE	SALESMAN	CUSTOMER	PRODUCT	AMOUNT
1	02	4	2	1	20.05
2	02	2	4	1	40.00
3	04	1	17	2	100.00
4	04	6	14	1	55.20
5	06	3	2	2	45.00
6	08	5	1	2	33.24

Must be resorted into the following table:

ROW	DATE	SALESMAN	CUSTOMER	PRODUCT	AMOUNT
				1	
				1	
				1	
				2	
				2	
				2	

The use of a separate **CHAIN** array makes it possible to avoid the cumbersome process of repositioning the entire matrix. Instead, an array of pointers indicates the order in which the rows have to be listed to achieve the correct sequence.

The simple program listed below shows how this is done:

```
10 LET C(4) = 2

20 LET C(2) = 1

30 LET C(1) = 6

40 LET C(6) = 5

50 LET C(5) = 3

60 LET C(3) = 0

70 LET H = 4

80 IF H = 0 THEN 120

90 FOR M = 1 TO 5: PRINT R$(H,M): NEXT M

100 LET H = C(H)

110 GOTO 80

120 END
```

The chain array is defined in Lines 10 to 60. It is then used to extract the appropriate rows of matrix **R$(,)** by a process which begins at Line 70. The first entry of **H**, the head of the chain, starts the sequence. At Line 100, each value of **C( )** 'points' to the next row to be printed. **C(4)** points at row **2**; **C(2)** points at row **1** and so on. Chaining is terminated when the zero value of **C(3)**, the tail of the chain, is detected in Line 80. This program forms the basis of the printing routine used in **WHO'S** from Line 3116.

How then is such a chain created? Look at column **4** of matrix **RE$(,)**. The first element of the chain array, the head, must refer to one of the rows containing Product 1. That is either rows 1, 2 or 4. These rows can be identified by a simple search through the column using the program listed below:

```
10 FOR R = 1 TO 6

20 IF R$(R,4) = "1" THEN H = R

30 NEXT R

40 PRINT H
```

When the program has finished running, the value of **H** will be the last row in the matrix which contains a **1** in column **4**, that is, row **4**. This fixes the position of the head of the chain. The first link in the chain is the value of **C(4)** which is set equal to (i.e. pointed to) the number of the next row in order. This row is either another "**1**" row or, failing that, the next highest value. Provided there is another **1** entry in the column, it will be found between rows **1** and **3** as follows:

```
10 FOR R = 1 TO 3

20 IF RE$(R,4) = "1" THEN C(4) = R

30 NEXT R

40 PRINT C(4)
```

The result is **C(4)=2** and if the process is repeated once more for **R=1** to **2**, then **C(2)=1**. This exhausts the "**1**" entries in the list. When the program is run again **C(1)** takes a zero value. Now look for the next highest value in column **3**, in this case **2**. Running the program with this new value, the head of the chain is **6, C(6)=5**, **C(5)=3** and **C(3)** is zero. The full result is as follows:

LIST "1"	LIST "2"
H = 4	H = 6
C(4) = 2	C(6) = 5
C(2) = 1	C(5) = 3
C(1) = 0	C(3) = 0

To create a single array listing the entire column, the **C(1)** link is set equal to the **H** of list **2** (just like joining a bicycle chain) with the following result:

or in proper order:

H = 4	H = 4
C(4) = 2	C(1) = 6
C(2) = 1	C(2) = 1
C(1) = 6	C(3) = 0
C(6) = 5	C(4) = 2
C(5) = 3	C(5) = 3
C(3) = 0	C(6) = 5

This is the original sequence that was used to create the correct order of **R$(,)** at the start of the section.

However, the earlier arrangement of **2** chains, each terminating in a zero entry, is more useful since it provides a convenient way of signalling that one category has finished and another is about to start. The link between the two chains can be made as follows:

```
IF C( ) = 0 THEN H = 6: REM THE NEW HEAD VALUE
```

Finally, it is convenient to incorporate the information on the heads of the chains in the chain arrays themselves rather than to define them separately. To do this, the subscripts are rearranged so that the head of each chain is assigned to one of the first elements of the array. Thus in this case, **C(1)** and **C(2)** hold the heads of the two chains and the rest of the array is assigned to **C(3)** through to **C(8)**. Similarly, **RE$(,)** now begins with two empty rows and the data lies between rows **3** and **8**.

The full program for generating the chains is shown below:

```
10 FOR N = 1 TO 2 : REM CATEGORIES "1" AND "2"

20 LET C(N) = 0

30 NEXT N

40 FOR N = 3 TO 8 : REM THE NUMBER OF ROWS

50 FOR L = 1 TO 2 : REM NUMBER OF CATEGORIES

60 IF VAL(RE$(N,4) = L THEN 80: REM COLUMN 4 IS
BEING SORTED

70 NEXT L

80 LET C(N) = C(L)

90 LET C(L) = N

100 NEXT N

110 FOR N = 1 TO 8: PRINT C(N): NEXT N
```

As a result, chain array **C( )** has the following values:

C(1) = 6

C(2) = 8

C(3) = 0

C(4) = 3

C(5) = 0

C(6) = 4

C(7) = 5

C(8) = 7

C(1) gives the head of the first chain as row **6**. The pointer sequence is **6** to **4** to **3** to **0** (the end of the first chain).

C(2) gives the head of the second chain as row **8**. Row **8** points to **7, 7** to **5** and **5** to **0** (the end of the second chain). Adjusting for the subscript change described earlier, it will be seen that this sequence ranks the original matrix **R$(,)** in product code order. This program forms the basis for **WHO'S** subroutine 8600.

To alter the position of one link in the chain, three array elements must be repositioned.

To illustrate this, return to the "**1**" category chain generated earlier and reproduced below as the column headed **ORIGINAL**. To change the position of the second entry, it is necessary to exchange the values of **C(4)** and **C(2)** and then to make **C(2)** return to the chain by pointing it at **6**. The result is shown below in the column marked **NEW**.

NEW	**ORIGINAL**
H = 4	**H = 4**
C(4) = 1	**C(4) = 2**
C(1) = 2	**C(2) = 1**

C(2) = 6	**C(1) = 6**
C(6) = 5	**C(6) = 5**
C(5) = 3	**C(5) = 3**
C(3) = 0	**C(3) = 0**

When these same arrays are put into their correct subscript sequence, they look like this:

NEW	**ORIGINAL**
H = 4	**H = 4**
C(1) = 2	**C(1) = 6**
C(2) = 6	**C(2) = 1**
C(3) = 0	**C(3) = 0**
C(4) = 1	**C(4) = 2**
C(5) = 3	**C(5) = 3**
C(6) = 5	**C(6) = 5**

Put in more general terms, the sequence for exchanging two rows (1 and 2) is as follows:

NEW	**ORIGINAL**
H (START LINK)	**H**
C(H) = 1	**C(H) = 2**
C(1) = 2	**C(2) = 1**
C(2) = F (FINISH LINK)	**C(1) = F**

And the exchange is accomplished by the following program lines:

```
10 REM H IS THE START LINK AND F THE FINISH LINK.

20 LET P = H

30 LET H = C(H)

40 LET F = C(H)

50 LET X = C(P) : REM X HOLDS THE VALUE WHILST
THE CHANGEOVER IS MADE.

60 LET C(P) = C(H)

70 LET C(H) = C(F)

80 LET C(F) = X

90 LET C = F
```

This program forms part of the bubblesort in **WHO'S** subroutine 8800.

Returning now to the main **WHO'S** program at Line 3058, the operator is asked to specify two fields on which the data are to be analysed. Flag **FL** controls the order in which the two fields are entered. Any two fields can be chosen, for example, a first sort into products followed by a sort into customers. The operator enters the two fields in order and they become the values of **M1** and **M2** in Lines 3086 and 3088. The only restriction is that the date cannot be chosen as the first sorting field (Line 3066).

The next stage is to sort the whole of **R$(,)** on the first field, **M1**, in subroutine 8600. However, before it is called, subroutine 8600 must be 'told' how many rows it will have to sort and how many categories they are to be sorted into. The first of these parameters is **NT**, the total of the individual transactions read in from the sales data files and the second is the number of entries in the directory which relates to the first field chosen for sorting. This latter number **(NR)** is extracted from the directory file using Subroutine 5000

Subroutine 8600 has already been described in detail. Note that in the full program listing, the subscripts of the chain array, **C( )**, are arranged so that they 'offset' the data matrix **R$(,)** by **NR** places. This device enables the first **NR** elements in **C( )** to be used to store the heads of each chain (**NR** of them) without having to leave an equivalent number of rows of **R$(,)** blank. Note also the counter, **N( )**, in Line 8612 which records the number of items in each category in preparation for the bubblesort which follows.

On exiting from the first sort, the program moves immediately into the second sort on the field defined by **M2**. This is a bubblesort within each of the **NR** categories established in the first sort. It takes place in subroutine 8800 which is called **NR** times, each time with a different chain head being entered as the variable **R**.

Having completed the second sort, the final stage is to print the output by applying the chain array to matrix **RE$(,)**. This is done from Line 3118 onwards, using the chain array to list out **R$(,)** as was described earlier in the chapter. Variable **A** handles screen scrolling by counting the number of lines which are printed and the **N** loop counts through the **NR** categories into which the data are now sorted. Once a screenful of data has been printed, control is returned to the operator at Line 3136. He presses the space bar to initiate another circuit of the printing routine.

Delete redundant data:

Option (3) calls subroutine 4000 which is very similar to the earlier part of the analysis routine. The time period covering the files to be deleted is specified, they are identified and then deleted.

```
100 REM WHO'S SELLING WHAT
101 CLOSE 8
102 LET NS=20
104 LET NC=50
106 LET NP=20
108 LET FL=0
110 DIM N(20)
114 DIM R$(100,5),B$(50),C(150),S$(20),C
$(50),P$(20)
116 LET DE=0
118 CD$="
"
120 CR$="
"
1000 REM MAIN MENU
1002 PRINT"
"
1004 PRINT "     WHO'S SELLING WHAT":PRI
NT"     ================="
1006 PRINT "          1) ENTER A SALES R
ECORD":PRINT"2) ANALYSE SALES"
1007 PRINT"3) DELETE REDUNDANT DATA":PRI
NT"4) ENTER NAMES IN DIRECTORIES"
1008 PRINT"5) SET UP A NEW DATABASE":PRI
NT"6) QUIT"
1009 PRINT"   ENTER OPTION NUMBER   ";:
LET D=1:GOSUB 7000
1010 IF RF=1 THEN GOTO 1017
1011 IF RF=2 THEN GOTO 1024
1012 IF RF=3 THEN GOTO 1016
1013 LET I$="ENTER 5 IF THIS IS THE FIRS
T RUN":GOSUB 9002
1014 GOTO 1000
1015 GOSUB 9502
1016 GOTO 1008
1017 IF TV=1 THEN GOSUB 1500
1018 IF TV=2 THEN GOSUB 3000
1019 IF TV=3 THEN GOSUB 4000
1020 IF TV=4 THEN GOSUB 6500
1021 IF TV=5 THEN GOSUB 6000
1022 IF TV=6 THEN GOSUB 9500
```

```
1023 GOTO 1000
1024 PRINT "ENTRY
OUTSIDE RANGE - TRY AGAIN":GOTO 1008
1500 LET FL=0
1501 IF FL=1 THEN GOTO 1506
1502 IF FL=2 THEN GOTO 1508
1504 LET D$="SADIR":GOTO 1514
1506 LET D$="CUDIR":GOTO 1514
1508 LET D$="PRDIR":GOTO 1514
1510 REM ENTER AN EXPENSE ITEM
1514 OPEN 8,8,8,"0:"+D$+"S,R"
1518 INPUT#8,NR:LET N(FL+1)=NR
1520 FOR N=1 TO NR
1522 IF FL=1 THEN GOTO 1526
1523 IF FL=2 THEN GOTO 1528
1524 INPUT#8,S$(N):GOTO 1530
1526 INPUT#8,C$(N):GOTO 1530
1528 INPUT#8,P$(N)
1530 NEXT N
1532 CLOSE 8
1534 IF FL=2 THEN GOTO 1538
1536 LET FL=FL+1:GOTO 1501
1538 REM ENTER AN EXPENSE ITEM
1540 PRINT"IS THIS THE FIRST ENTRY THIS
 MONTH ?":GOSUB 8500
1541 IF RF=1 THEN GOTO 1552
1543 IF RF=3 THEN GOTO 1548
1544 LET I$="IF SO A NEW FILE MUST BE MA
DE  ENTER Y":GOSUB 9000
1546 GOTO 1540
1548 GOSUB 9500
1550 GOTO 1540
1552 IF TV=1 THEN LET FL=1
1554 REM ENTER AN EXPENSE ITEM
1556 REM ENTER DATE
1558 PRINT"ENTER THE TRANSACTION DATE":
PRINT"========================="
1560 GOSUB 8700
1562 PRINT"DAY (1 TO 31)
   ";:LET D=2:GOSUB 7000
```

```
1563 IF RF=1 THEN GOTO 1574
1564 IF RF=2 THEN GOTO 1576
1565 IF RF=3 THEN GOTO 1570
1566 I$="THIS DATE WILL BE RECORDED ON T
HE SALES RECORD":GOSUB 9002
1568 GOTO 1562
1570 GOSUB 9502
1572 GOTO 1562
1574 PRINT"========
================================="
1576 IF TV<1 OR TV>31 THEN PRINT"
NUMBER OUTSIDE RANGE"
1577 IF TV<1 OR TV>31 THEN GOTO 1562
1578 IF LEN(T$)=1 THEN T$="0"+T$
1580 LET Y$=T$:REM DAY
1582 IF FL=1 THEN GOSUB 8900:LET FL=0
1586 REM SALES RECORD ENTRY
1588 LET FY=0:LET A=20
1590 FOR M=1 TO A
1592 LET R$(M,1)=Y$
1594 PRINT"SALES RECORD":PRINT"========
===="
1596 PRINT"SALESMAN NUMBER
 ";
1598 IF FY<>1 THEN GOTO 1600
1599 D=2:LET T$=STR$(VAL(R$(M,2))):LET C
H=LEN(T$):PRINT T$;:GOSUB 7004:GOTO 1602
1600 LET D=2:GOSUB 7000
1601 IF RF=1 THEN GOTO 1616
1602 IF RF=2 THEN GOTO 1614
1603 IF RF=3 THEN GOTO 1610
1604 IF FY=1 THEN GOTO 1616
1606 I$="A NUMBER BETWEEN 1 AND "+STR$(N
S):GOSUB 9000
1608 GOTO 1596
1610 GOSUB 9502
1612 GOTO 1596
1614 PRINT"ENTRY OUTS
IDE RANGE - TRY AGAIN":GOTO 1596
```

```
1616 PRINT""
1618 IF TV>N(1) THEN GOTO 1614
1620 PRINT"";S$(TV):LET R$(M,2)=T$
1622 REM R$(M,2) IS SALESMAN NUMBER
1624 PRINT"GROSS SALES ";
1625 IF FY<>1 THEN GOTO 1628
1626 T$=STR$(VAL(R$(M,5))):CH=LEN(T$):PRINT T$;:DE=1:D=7:GOSUB 7004:GOTO1630
1628 LET D=7:LET DE=1:GOSUB 7000
1629 IF RF=1 THEN GOTO 1644
1630 IF RF=2 THEN GOTO 1642
1631 IF RF=3 THEN GOTO 1638
1632 IF FY=1 THEN GOTO 1644
1634 LET I$="SALES AMOUNT 7 CHARACTERS MAX.":GOSUB 9000
1636 GOTO 1624
1638 GOSUB 9502
1640 GOTO 1624
1642 PRINT"ENTRY OUTSIDE RANGE - TRY AGAIN":GOSUB 9000
1644 LET DE=0:PRINT""
1646 R$(M,5)=T$:REM GROSS AMOUNT
1648 PRINT"CUSTOMER NO. ";
1650 IF FY<>1 THEN GOTO 1652
1651 T$=STR$(VAL(R$(M,3))):CH=LEN(T$):PRINT T$;:LET D=2:GOSUB 7004:GOTO 1654
1652 LET D=2:GOSUB 7000
1653 IF RF=1 THEN GOTO 1668
1654 IF RF=2 THEN GOTO 1666
1655 IF RF=3 THEN GOTO 1662
1656 IF FY=1 THEN GOTO 1668
1658 I$="A NUMBER BETWEEN 1 AND "+STR$(NC):GOSUB 9000
1660 GOTO 1648
```

```
1662 GOSUB 9502
1664 GOTO 1648
1666 PRINT"ENTRY OUTSIDE RANGE - TRY AGAIN":GOTO 1648
1668 PRINT""
1670 IF TV>N(2) THEN GOTO 1666
1672 PRINT"";C$(TV):LET R$(M,3)=T$
1674 REM R$(M,3) IS CUSTOMER NUMBER
1676 PRINT"PRODUCT NUMBER ";
1678 IF FY<>1 THEN GOTO 1680
1679 T$=STR$(VAL(R$(M,4))):CH=LEN(T$):D=2:PRINT T$;:GOSUB 7004:GOTO 1681
1680 GOSUB 7000
1681 IF RF=1 THEN GOTO 1696
1682 IF RF=2 THEN GOTO 1692
1683 IF RF=3 THEN GOTO 1690
1684 IF FY=1 THEN GOTO 1696
1686 I$="A NUMBER BETWEEN 1 AND "+STR$(NP):GOSUB 9000
1688 GOTO 1676
1690 GOSUB 9502
1692 GOTO 1676
1694 PRINT"ENTRY OUTSIDE RANGE - TRY AGAIN":GOTO 1676
1696 PRINT""
1698 IF TV>N(3) THEN GOTO 1694
1700 PRINT"";P$(TV):LET R$(M,4)=T$
1702 REM R$(M,4) IS PRODUCT NUMBER
1704 IF FY=1 THEN GOTO 1720
1706 PRINT"WANT TO ENTER ANOTHER SALE ":GOSUB 8500
1707 IF RF=1 THEN GOTO 1717
1708 IF RF=2 THEN GOTO 1708
1709 IF RF=3 THEN GOTO 1714
```

```
1710 I$="IF N YOU WILL SAVE DATA AND RET
URN TO   THE MENU":GOSUB 9000
1712 GOTO 1706
1714 GOSUB 9502
1716 GOTO 1706
1717 IF TV=1 THEN GOTO 1720
1718 IF TV=2 THEN NE=M:M=A
1720 NEXT M
1724 PRINT"▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒▒DO YOU W
ANT TO CORRECT THE ENTRIES ";
1726 GOSUB 8500
1727 IF RF=1 THEN GOTO 1738
1729 IF RF=3 THEN GOTO 1734
1730 I$="IF Y THEN YOU WILL GET THE LIST
 AGAIN":GOSUB 9000
1732 GOTO 1724
1734 GOSUB 9500
1736 GOTO 1724
1738 IF TV=1 THEN FY=1:A=NE:PRINT"▒IF O.
K. THEN RETURN":GOTO 1590
2000 REM MAKE FILE ENTRIES
2006 OPEN 8,8,8,"0:"+W$+Z$+"S,R"
2010 INPUT#8,NR
2012 IF NR=-999999 THEN NR=0:NT=NE:GOTO
2028
2014 NT=NR+NE
2016 N=NE+1
2019 FOR M=1 TO 5
2020 INPUT#8,R$(N,M)
2022 NEXT M
2024 IF N=NT THEN GOTO 2028
2026 N=N+1:GOTO 2019
2028 CLOSE 8
2030 OPEN 8,8,8,"@0:"+W$+Z$+"S,W"
2034 PRINT#8,NT
2036 N=1
2038 FOR M=1 TO 5
2040 PRINT#8,R$(N,M)
2042 NEXT M
```

```
2044 IF N=NT THEN GOTO 2048
2046 N=N+1:GOTO 2038
2048 CLOSE 8
2050 GOTO 1002
2052 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■THIS IS
THE FIRST ENTRY THIS MONTH■";
2054 GOTO 1004
3000 REM SALES ANALYSIS
3002 REM ENTER DATE
3004 LET FL=0
3006 PRINT"■":IF FL=1 THEN PRINT"ENTER T
HE FINISH DATE":GOTO 3009
3008 PRINT"ENTER THE START DATE"
3009 PRINT"====================="
3010 GOSUB 8700
3012 IF FL=1 THEN GOTO 3016
3014 V$=W$:G$=Z$:FL=1:GOTO 3006
3016 X$=W$:H$=Z$:FL=0
3018 REM LOAD THE CHOSEN FILES
3020 T=VAL(V$):D=VAL(G$)
3022 NE=1:NT=0:W$=V$:Z$=G$
3024 OPEN 8,8,8,"0:"+W$+Z$+"S,R"
3028 PRINT"■■■■■■■■■■■■■■■■■■DATA IS BEING
 LOADED"
3030 INPUT#8,NR
3032 NT=NT+NR
3034 FOR N=1 TO NR
3036 FOR M=1 TO 5
3038 INPUT#8,R$(N,M)
3040 NEXT M
3042 R$(N,1)=W$+Z$+R$(N,1)
3044 NEXT N
3046 CLOSE 8
3048 NE=NT+1
3050 D=D+1:IF D>12 THEN T=T+1:D=1
3052 W$=MID$(STR$(T),2,3):Z$=MID$(STR$(D
),2,3):IF LEN(Z$)=1 THEN Z$="0"+Z$
3054 IF W$=X$ AND Z$=H$ THEN GOTO 3058
3056 GOTO 3024
3058 REM SORT ON CATEGORY
```

```
3060 REM ENTER CATEGORY AND NUMBER OF EN
TRIES IN CAT.
3062 PRINT"CHOOSE PRIMARY SORTING CATEG
ORY":IF FL=0 THEN GOTO 3066
3064 PRINT"CHOOSE SECONDARY SORT CATEGO
RY":GOTO 3068
3066 PRINT"DO NOT E
NTER 1 FOR PRIMARY FIELD"
3068 PRINT"1) DATE":PRINT"2) SALESMA
N":PRINT"3) CUSTOMER":PRINT"4) PRODUCT"
3070 PRINT"ENTER SORTING FIEL
D ?     ";
3072 D=1:GOSUB 7000
3073 IF RF=1 THEN GOTO 3084
3074 IF RF=2 THEN GOTO 3090
3075 IF RF=3 THEN GOTO 3080
3076 I$="THIS DETERMINES THE WAY THE LIS
T IS      ORDERED":GOSUB 9000
3078 GOTO 3062
3080 GOSUB 9500
3082 GOTO 3062
3084 PRINT"
                                       "
3085 PRINT"
               "
3086 IF FL=0 THEN M1=TV:FL=1:GOTO 3064
3088 M2=TV:FL=0:GOTO 3092
3090 PRINT"ILLEGAL NUMBER - REENTER":FL=
0:GOTO 3062
3092 M=M1:REM FIRST SORT
3093 IF M=1 THEN GOTO 3090
3094 IF M=3 THEN GOTO 3098
3095 IF M=4 THEN GOTO 3100
3096 D$="SADIR":GOSUB 5000:GOTO 3102
3098 D$="CUDIR":GOSUB 5000:GOTO 3102
3100 D$="PRDIR":GOSUB 5000
3102 PRINT"DATA IS BEING
 SORTED FOR PRINTING"
3104 GOSUB 8600
3106 REM SORT ON SECOND CATEGORY
```

```
3108 M=M2
3110 FOR N=1 TO NR
3112 GOSUB 8800
3114 NEXT N
3116 PRINT"■"
3118 PRINT"DATE  S'MAN CUST PROD SALES"
3120 A=0
3122 FOR N=1 TO NR
3126 R=C(N)
3128 IF R=0 THEN GOTO 3156
3130 T$=R$(R-NR,1):PRINT LEFT$(T$,6);"
";:FOR L=2 TO 4:T$=R$(R-NR,L)
3131 PRINT T$;"     ";:NEXT L:T$=R$(R-NR,
5):PRINT LEFT$(T$,8)
3132 PRINT:A=A+1
3134 IF A<10 THEN GOTO 3150
3136 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■PRESS SP
ACE BAR TO CONTINUE":GOSUB 8000
3138 IF RF=1 THEN GOTO 3148
3139 IF RF=3 THEN GOTO 3144
3140 I$="THE SCREEN WILL FILL UP WITH TH
E NEXT   PART OF THE LISTING":GOSUB 9000
3142 GOTO 3136
3144 GOSUB 9500
3146 GOTO 3136
3148 A=0:PRINT"■"
3150 R=C(R)
3152 IF R=0 THEN GOTO 3156
3154 GOTO 3128
3156 A=A+1:NEXT N
3158 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■WANT ANOTH
ER ANALYSIS Y/N ?":GOSUB 8500
3160 IF RF=1 THEN GOTO 3170
3161 IF RF=3 THEN GOTO 3166
3162 I$="IF Y YOU CAN SELECT TWO MORE FI
ELDS ,   IF N YOU RETURN":GOSUB 9000
3164 GOTO 3158
3166 GOSUB 9500
3168 GOTO 3158
```

```
3170 IF TV=2 THEN GOTO 1002
3171 IF TV=1 THEN GOTO 3174
3172 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■THERE IS S
OMETHING WRONG WITH THE DATA":GOTO 1004
3174 GOTO 3000
4000 REM DELETE REDUNDANT DATA
4002 REM ENTER DATE
4004 LET FL=0
4006 PRINT"■":IF FL=0 THEN PRINT"CHOOSE
DATE TO DELETE FROM":GOTO 4008
4007 PRINT"CHOOSE DATE TO DELETE TO"
4008 PRINT"==============================
======"
4010 GOSUB 8700
4012 IF FL=1 THEN GOTO 4016
4014 V$=W$:G$=Z$:FL=1:GOTO 4006
4016 X$=W$:H$=Z$:FL=0
4018 REM DELETE THE SELECTED FILES
4020 T=VAL(V$):D=VAL(G$)
4022 NE=1:NT=0:W$=V$:Z$=G$
4024 OPEN 15,8,15
4025 PRINT#15,"SCRATCH0:"+W$+Z$
4026 CLOSE 15
4028 NE=NT+1
4030 D=D+1:IF D>12 THEN T=T+1:D=1
4032 W$=MID$(STR$(T),2,3):Z$=MID$(STR$(D
),2,3):IF LEN(Z$)=1 THEN Z$="0"+Z$
4034 IF W$=X$ AND Z$=H$ THEN GOTO 4038
4036 GOTO 4024
4038 GOTO 1002
5000 REM LOOK IN FILE TO FIND NAME
5002 PRINT " ";
5004 OPEN 8,8,8,"0:"+D$
5008 INPUT#8,NR
5016 CLOSE 8
5018 RETURN
6000 REM SET UP A NEW DISK
6002 PRINT"■ANY FILES ON THE DISK WILL B
E DESTROYED"
```

```
6004 PRINT"■THE ASSUMPTION IS THAT YOU A
RE STARTING AGAIN WITH NEW DATA"
6005 PRINT"DO YOU WANT TO CONTINUE ?"
6006 GOSUB 8500
6007 IF RF=1 THEN GOTO 6018
6008 IF RF=3 THEN GOTO 6014
6010 I$="THE ASSUMPTION IS THAT YOU ARE
STARTING AGAIN WITH NEW DATA":GOSUB 9000
6012 GOTO 6002
6014 GOSUB 9500
6016 GOTO 6002
6018 IF TV=1 THEN GOTO 6022
6019 IF TV=2 THEN GOTO 1002
6020 GOTO 6002
6022 FL=0
6023 IF FL=1 THEN GOTO 6028
6024 IF FL=2 THEN GOTO 6030
6025 IF FL=3 THEN GOTO 6048
6026 D$="SADIR":GOTO 6034
6028 D$="CUDIR":GOTO 6034
6030 D$="PRDIR"
6034 OPEN8,8,8,"@0:"+D$+"S,W"
6040 PRINT#8,-999999
6042 CLOSE 8
6044 FL=FL+1
6046 GOTO 6023
6048 GOTO 1002
6500 REM ENTER RECORDS IN DIRECTORY
6501 FL=0
6502 PRINT"■":FY=0
6504 D$="SADIR":J$="S/MAN NO.   ":NT=NS
6506 PRINT
6508 OPEN8,8,8,"0:"+D$+"S,R"
6512 INPUT#8,NR
6514 IF NR=-999999 THEN NR=0:GOTO 6522
6516 FOR N=1 TO NR
6518 INPUT#8,B$(N)
6520 NEXT N
6522 CLOSE 8
6524 IF NR>0 THEN FY=1
6526 PRINT"■RECORD ENTRY":PRINT"========
====="
6528 PRINT"( PRESS RETURN FOR NEXT ENTRY
 )"
```

```
6529 PRINT"( WHEN FINISHED PRESS * TO ED
IT OR . TO   CONTINUE )"
6530 FOR J=1 TO NT
6532 IF J=1 THEN GOTO 6540
6534 FOR N=21-J TO 19
6536 IF N>0 THEN PRINT"■";LEFT$(CD$,N);"
                                        ";
6537 IF N>0 THEN PRINT"■";LEFT$(CD$,N);"
▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐";B$(J-(20-N))
6538 NEXT N
6540 PRINT"■";LEFT$(CD$,20);J$;J:PRINT"■
";LEFT$(CD$,20);"▐▐▐▐▐▐▐▐▐▐▐▐▐▐▐";
6541 PRINT"                ▐▐▐▐▐▐▐▐▐▐▐▐▐";
6542 IF FY=1 AND J<NR+1 THEN T$=B$(J):PR
INTT$;:D=9:CH=LEN(T$)
6543 IF FY=1 AND J<NR+1 THEN GOSUB 8554:
GOTO 6547
6544 FY=0:NR=NT
6546 D=9:GOSUB 8550
6547 IF RF=1 THEN GOTO 6562
6548 IF RF=2 THEN GOTO 6560
6549 IF RF=3 THEN GOTO 6556
6550 IF RF=4 THEN GOTO 6570
6551 IF RF=5 THEN GOTO 6574
6552 IF FY=1 THEN GOTO 6562
6553 I$="ENTER NAME (UP TO 9 CHARACTERS)
":GOSUB 9000
6554 GOTO6540
6556 GOSUB 9500
6558 GOTO 6540
6560 PRINT"■";LEFT$(CD$,17);"ENTRY LONGE
R THAN 9 CHARS":GOTO 6532
6562 PRINT"■";LEFT$(CD$,17);"
                "
6564 B$(J)=T$
6566 NEXT J
6568 GOTO 6574
6570 GOSUB 6750
6572 GOTO 6532
6574 IF FY=1 THEN FY=0
6576 PRINT"□"
6578 PRINT"DO YOU WANT TO CORRECT THE EN
TRIES ?";
```

```
6580 GOSUB 8500
6582 IF RF=1 THEN GOTO 6592
6583 IF RF=3 THEN GOTO 6588
6584 I$="IF Y THEN YOU WILL GET THE LIST
 AGAIN.":GOSUB 9000
6586 GOTO 6574
6588 GOSUB 8500
6590 GOTO6574
6592 IF TV=2 THEN GOTO 6606
6594 REM CHECKING ROUTINE
6596 PRINT"▟"
6598 PRINT"ENTER NAME UP TO 9 CHARS":PRI
NT"========================"
6600 PRINT"IF O.K. THEN RETURN"
6602 PRINT"▣IF WRONG - REENTER"
6604 FY=1:GOTO 6530
6606 OPEN 8,8,8,"@0:"+D$+"S,W"
6610 PRINT#8,J-1
6612 FOR N=1 TO J
6614 PRINT#8,B$(N)
6616 NEXT N
6618 CLOSE 8
6620 IF FL=1 THEN GOTO 6624
6621 IF FL=2 THEN GOTO 6626
6622 FL=1:D$="CUDIR":J$="CUST. NO.":NT=N
C:GOTO 6506
6624 FL=2:D$="PRDIR":J$="PROD. NO.":NT=N
P:GOTO 6506
6626 GOTO 1002
6750 REM EDIT ROUTINE
6752 L=J: IF L>19 THEN L=19
6756 PRINT"▣";LEFT$(CD$,20-L);"▮▮▮▮▮▮▮▮▮
▮▮▮▮▮▮▮▮▮▮▮▮▮▮CORRECTION"
6758 PRINT"▣";LEFT$(CD$,21-L);"▮▮▮▮▮▮▮▮▮
▮▮▮▮▮▮▮▮▮▮▮▮▮▮";:D=9:GOSUB 8550
6760 IF RF=1 THEN GOTO 6766
6761 IF RF=2 THEN GOTO 6772
6762 IF RF=3 THEN GOSUB 9500:GOTO 6774
6764 L=L-1:IF L<=1 THEN GOTO 6774
6765 GOTO 6758
6766 PRINT"▣▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▯▮";
6768 B$(J-L+1)=T$:GOTO 6764
6770 REM
6772 PRINT"▣";LEFT$(CD$,22);"TYPE THE NA
MES, UP TO 9 CHARS":GOTO 6758
```

```
6774 PRINT"■";LEFT$(CD$,20-L);:RETURN
7000 REM NUMBER ENTRY
7002 TV=0:T$="":CH=0
7004 GET A$:IF A$<>"" THEN GOTO 7004
7005 GET A$:IF A$="" THEN GOTO 7005
7006 IF ASC(A$)=13 AND CH=0 THEN RF=0:GO
TO 7044
7010 IF ASC(A$)=20 THEN GOTO 7032
7012 IF ASC(A$)=13 THEN RF=1:GOTO 7042
7014 IF A$="@" AND CH=0 THEN RF=3:GOTO 7
044
7016 IF CH>=D THEN RF=2:GOTO 7044
7018 IF A$="-" AND CH=0 THEN RF=1:GOTO 7
028
7020 IF A$="." AND DE=1 THEN GOTO 7028
7022 IF A$>="0" AND A$<="9" THEN GOTO 70
28
7026 GOTO 7004
7028 T$=T$+A$:PRINT A$;:CH=CH+1
7030 GOTO 7004
7032 IF CH=0 THEN GOTO 7004
7034 CH=CH-1:PRINT"■■ ■■";
7036 IF CH=0 THEN T$="":GOTO 7004
7038 T$=LEFT$(T$,LEN(T$)-1)
7040 GOTO 7004
7042 TV=VAL(T$)
7044 RETURN
8000 REM SPACE BAR INPUT
8002 TV=0:T$="":CH=0
8004 GET A$:IF A$=" " THEN GOTO 8004
8006 GET A$:IF A$<>" " THEN GOTO 8006
8008 RF=1:RETURN
8500 REM Y OR N INPUT
8502 TV=0:T$="":CH=0
8504 GET A$:IF A$<>"" THEN GOTO 8504
8505 GET A$:IF A$=""THEN GOTO 8505
8506 IF ASC(A$)=13 AND CH=0 THEN RF=0:GO
TO 8538
8508 IF A$="@" AND CH=0 THEN RF=3:GOTO 8
538
8510 IF ASC(A$)=20 THEN GOTO 8526
8512 IF ASC(A$)=13 THEN RF=1:GOTO 8536
8516 IF A$="Y" AND CH=0 THEN T$="1":RF=1
:GOTO 8522
8518 IF A$="N" AND CH=0 THEN T$="2":RF=2
:GOTO 8522
```

```
8520 GOTO 8504
8522 PRINT A$;:LET CH=CH+1
8524 GOTO 8504
8526 IF CH=0 THEN GOTO 8504
8528 CH=CH-1:PRINT"■■ ■■";
8530 IF CH=0 THEN T$="":GOTO 8504
8532 T$=LEFT$(T$,LEN(T$)-1)
8534 GOTO 8504
8536 TV=VAL(T$)
8538 RETURN
8550 REM LETTER ENTRY
8552 TV=0:T$="":CH=0
8554 GET A$:IF A$<>"" THEN GOTO 8554
8555 GET A$:IF A$="" THEN GOTO 8555
8556 IF A$="*" AND CH=0 THEN RF=4:GOTO 8
592
8558 IF ASC(A$)=13 AND CH=0 THEN RF=0:GO
TO 8592
8560 IF A$="." AND CH=0 THEN RF=5:GOTO 8
592
8562 IF A$="@" AND CH=0 THEN RF=3:GOTO 8
592
8564 IF ASC(A$)=20 THEN GOTO 8580
8566 IF ASC(A$)=13 THEN RF=1:GOTO 8592
8570 IF CH>0 THEN RF=2:GOTO 8592
8572 IF ASC(A$)>47 AND ASC(A$)<123 THEN
GOTO 8576
8574 GOTO 8554
8576 T$=T$+A$:PRINT A$;:CH=CH+1
8578 GOTO 8554
8580 IF CH=0 THEN GOTO 8554
8582 CH=CH-1:PRINT"■■ ■■";
8584 IF CH=0 THEN T$="":GOTO 8554
8586 T$=LEFT$(T$,LEN(T$)-1)
8588 GOTO 8554
8592 RETURN
8600 REM SORT ON CATEGORY
8602 FOR N=1 TO NR
8604 C(N)=0:N(N)=0
8606 NEXT N
8608 FOR N=NR+1 TO NT+NR
8610 L=1
8611 IF R$(N-NR,M)="" THEN GOTO 8614
8612 IF (VAL(R$(N-NR,M))=L) THEN N(L)=N(
L)+1:GOTO 8616
8614 L=L+1:IF L<=NR THEN GOTO 8611
```

```
8616 C(N)=C(L)
8618 C(L)=N
8620 NEXT N
8622 RETURN
8700 REM ENTER DATE
8702 PRINT"■■■■■■■■■■YEAR  ( TWO NUMBERS
)     ■■";:D=2:GOSUB 7000
8703 IF RF=1 THEN GOTO 8716
8704 IF RF=2 THEN GOTO 8714
8705 IF RF=3 THEN GOTO 8710
8706 I$="NUMBER BETWEEN 82 AND 90":GOSUB
 9002
8708 GOTO 8702
8710 GOSUB 9502
8712 GOTO 8702
8714 PRINT"■";LEFT$(CD$,20);"ENTRY OUTSI
DE RANGE":GOTO 8702
8716 PRINT"■";LEFT$(CD$,20);"
       "
8718 IF TV<82 OR TV>90 THEN PRINT"■";LEF
T$(CD$,20);"OUTSIDE RANGE":GOTO 8702
8720 W$=T$:REM YEAR
8722 PRINT"■■■■■■■■■■■MONTH ( TWO NUMBERS
 )     ■■";:D=2:GOSUB 7000
8723 IF RF=1 THEN GOTO 8734
8724 IF RF=2 THEN GOTO 8736
8725 IF RF=3 THEN GOTO 8730
8726 I$="NUMBER BETWEEN 1 AND 12":GOSUB
9002
8728 GOTO 8722
8730 GOSUB 9502
8732 GOTO 8722
8734 PRINT"■";LEFT$(CD$,20);"
                           "
8736 IF TV<1 OR TV>12 THEN PRINT"■";LEFT
$(CD$,20):"OUTSIDE RANGE":GOTO 8722
8738 Z$=T$:REM MONTH
8740 IF LEN(Z$)=1 THEN Z$="0"+Z$
8742 RETURN
8800 REM BUBBLESORT ON CATEGORY
8802 FOR S=1 TO N(N)-1
8804 Q=0
8806 R=N
8808 FOR L=1 TO N(N)-S
8810 P=R
8812 R=C(R)
```

```
8814 J=C(R)
8816 IF J=0 THEN GOTO 8838
8818 IF (VAL(R$(R-NR,M))<=VAL(R$(J-NR,M)
)) THEN GOTO 8832
8820 Q=1
8822 A=C(P)
8824 C(P)=C(R)
8826 C(R)=C(J)
8828 C(J)=A
8830 R=J
8832 NEXT L
8834 IF Q=0 THEN GOTO 8838
8836 NEXT S
8838 A=0:RETURN
8900 REM SETUP MAIN FILE
8902 PRINT"■"
8904 OPEN 8,8,8,"0:"+W$+Z$+"S,W"
8912 PRINT#8,-999999
8914 CLOSE 8
8916 RETURN
9000 REM HELP ROUTINE
9002 PRINT"■";LEFT$(CD$,18);"
                                    ";
9003 PRINT"
          ";
9004 PRINT"■";LEFT$(CD$,18);I$:PRINT"PRE
SS SPACE TO CONTINUE":GOSUB 8000
9006 IF RF=1 THEN GOTO 9014
9007 IF RF=3 THEN GOTO 9010
9008 GOTO 9002
9010 GOSUB 9502
9012 GOTO 9002
9014 PRINT"■";LEFT$(CD$,18);"
                                    ";
9016 PRINT"
          ";
9018 RETURN
9500 REM ESCAPE ROUTINE
9502 PRINT"■";LEFT$(CD$,20);"DO YOU WANT
 TO EXIT (Y/N) ?":GOSUB 8500
9504 IF RF=2 OR RF=3 THEN GOTO 9502
9506 IF TV=2 THEN GOTO 9510
9508 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■THE END■■
■■■■■■■■■=========":END
9510 PRINT"■":RETURN
```

9

SALESTREND
The Sales Manager's Package

Introduction:

The programs described in the previous chapters each tackle one aspect of sales management. However, for day to day use, it would clearly be preferable to have a single program, containing all these same features, which would cover all the sales manager's needs.

The precise arrangement of such a program depends very much on the role the manager occupies, the scope of his job and which aspects of it are the most important. This last chapter contains an outline description of how such a program can be built up from the elements which have already been described in earlier chapters.

The program is a comprehensive database which provides the basis of a system for monitoring performance in the market place. It makes use of the computer's inherently superior data handling facilities to create and maintain an ongoing record not just of total performance, but also the performance of the constituent elements of the business. The program also provides comprehensive facilities for data analysis which enable the manager to explore the underlying causes of changes in performance within his business.

Without the backup provided by a reliable and detailed database, it is near impossible to muster the sort of rational explanations and arguments neeeded to support management action and to provide convincing back-up to those vital reports 'up the line'. Furthermore, an up to date analysis of recent results must be to hand when subordinates' performance is being appraised since,

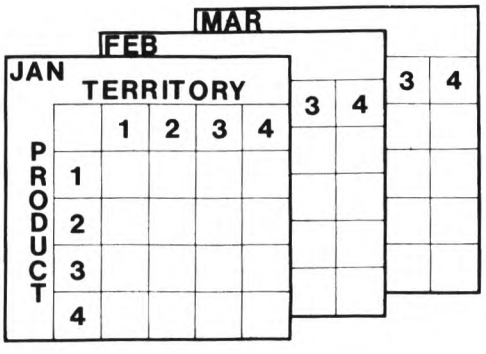
without such a framework, there is no limit to the number and ingenuity of the explanations which can be put forward to explain bad (or good) results.

Such a comprehensive system for logging and analysing data is beyond the scope of manual systems. Equally, it is normally not possible to achieve the required immediacy of access to up to date information from a system housed on a central mainframe computer. It is an ideal application for a personal computer.

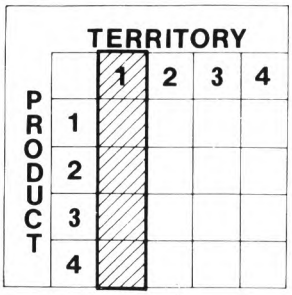
General description:

The database is held on file as a series of matrices, one for each period, which in this example, contain data on the performance of each product in each territory. Of course, the matrices could equally well be set up to hold data on the value of inquiries or of orders received or other regularily produced statistics.

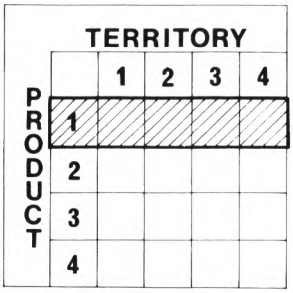
The basic scheme is illustrated below:



Each column contains the sales by product in a single territory.



Each row contains the sales by territory of a single product.



To examine the sales of a particular Product/Territory over time, the program works its way through the monthly records to create a new file. This file is made up of a sequence of sales statistics taken from the same cell of each monthly record.

Having assembled the basic data, the program has to present it in such a way that trends in the performance of the individual product/territory cells can be examined over time. This provides a powerful framework within which to track down the causes of differences in performance and the interrelationships between trends apparent in different parts of the business. The approach is illustrated in the following examples:

Suppose the performance of one particular product is giving concern. The first step would be to look at how total sales divide between territories. Now suppose that the poor performance turns out to be associated with only two territories. It looks as if the cause of the exceptional result is connected more with the two territories than with features which relate to the product itself. To confirm this proposition, it would be helpful to see how the exceptional territories had performed with other products and perhaps to compare their total sales with totals elsewhere.

A more difficult problem is seasonality. Difficult if only because it provides the rather hazy basis for remarks such as "We never sell much in January but it will pick up by April". Whether or not this view is reasonable, can be determined by examining past trends. If there is a seasonal pattern, then is it also likely to influence sales of other products and in other territories? If only some areas are seasonal, then is the characteristic associated with products or territories? And so on . . . . .

A purpose built marketing database is of enormous value in sales forecasting since its structure is an accurate reflection of the mechanisms by which sales are actually generated. The procedure for forecasting is to divide the business into its logical constitutents and to forecast each one forwards into the future. The total forecast is the sum of the constituents.

The elements of the program:

It will be evident that the main elements of the program have already been described in earlier chapters. **ADJUSTER** provides the means of taking account of inflation and the effect of working days. **GRAPHPLOTTER** covers the problem of scaling the graph and plotting a series of data points. **FORECASTER** contains the exponential smoothing model needed to project trends forward in to the future and finally, **WHO'S** gives a method of creating files and extracting selected periods for subsequent analysis. The main task remaining is how to fit the programs together within the confines of the computer.

However, before moving on, mention should be made of an important programming feature which has not been covered in earlier chapters, namely how to display data as a table. A 15 by 15 matrix is too large to fit onto the screen and so only part of it can be displayed at any one time. An ideal solution would be a 'spreadsheet' display which scrolls the screen in both the vertical and the horizontal planes. However, if this type of display is programmed in Basic it will be too slow for practical application. A less ambitious approach is to accept the computer's existing scrolling arrangements for the vertical plane and to gain something of the same spreadsheet effect by dividing the horizontal axis into sections. Under this scheme, the 15 by 15 matrix is divided into smaller matrices (say five 15 by 3 matrices), one of which is displayed in full at any one time. In addition to the 15 by 3 matrix, column and row totals, which are an important part of the data presentation, must also be accommodated along two sides of the display.

When the display first appears, matrix number **1** (plus totals) is displayed. To move on to number **2**, the operator presses the right arrow key and the matrix is printed. Another right arrow keystroke causes matrix number **3** to appear. To return to an earlier display, the operator presses the left arrow key. The display arrangements are controlled by two counters, one on the left arrow key and one on the right, both of which count the number of times their respective keys are pressed. The counters can take only the values **0, 1** or **2**. If the left arrow key is pressed when matrix number **1** is being displayed, that is with the left arrow counter at **0**, the counter remains at **0** and matrix number **1** is redisplayed. Similarly, at the other end of the display with the right arrow counter at **2**, another key press causes matrix number **2** to be displayed again and the counter remains at **2**. Other single key commands are provided to move on to another month's data and to exit from the data display on to the next part of the program.

Program structure:

Clearly there are many ways of arranging the detailed structure of a marketing database to suit the particular features of the business and the manager's requirements. However there are a number of

design issues and programming problems which are likely to appear regardless of the detailed configuration.

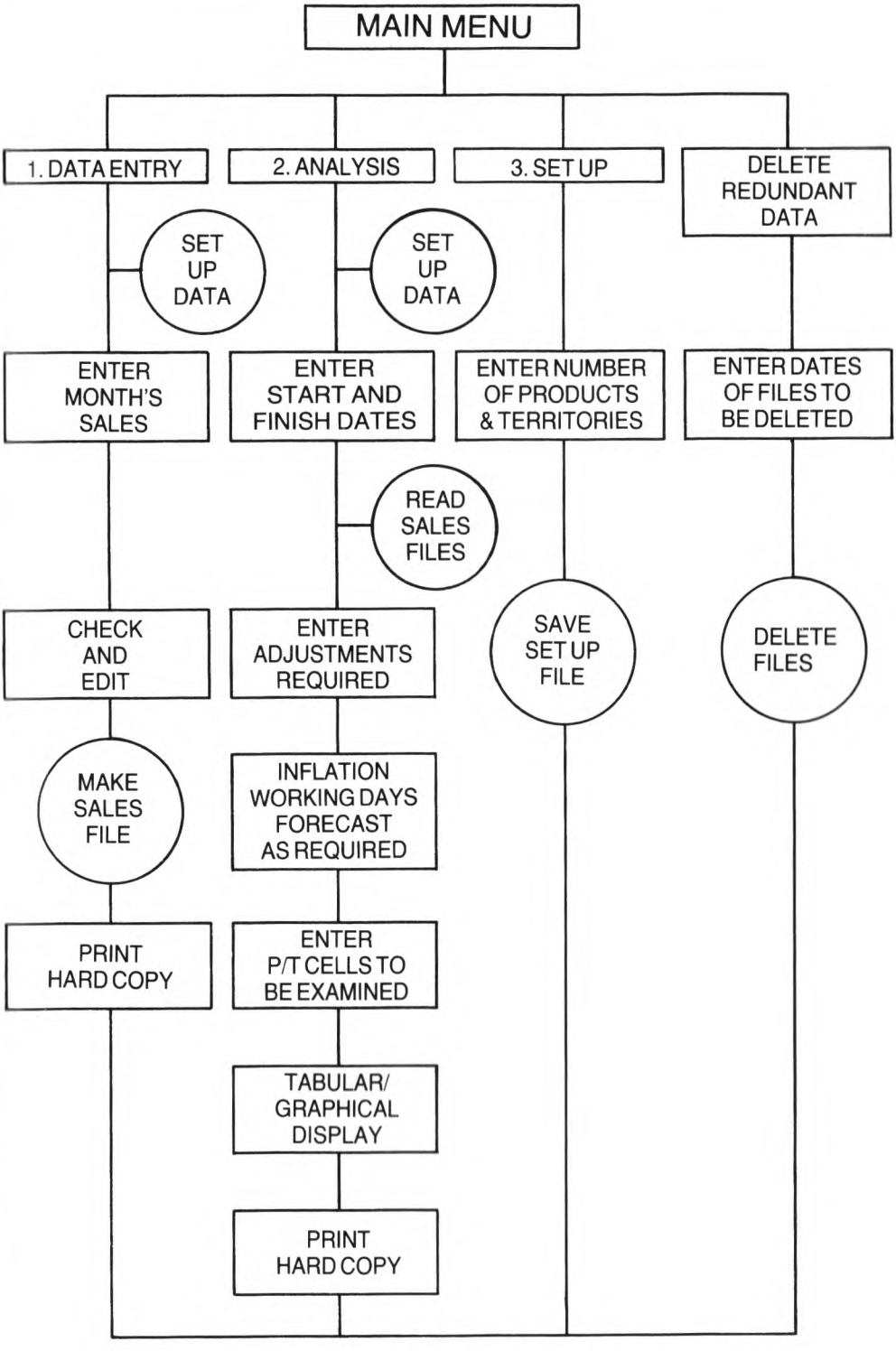
The program naturally divides into two main sections, data entry and data analysis. The first is relatively straightforward; it reads in the sales figures each month, positions them in the matrix, calculates totals and makes a data file. The second program is more involved. It has to extract data from the data files, analyse it, adjust for inflation, forecast and display the results either as a graph or as a table.

There should not be a problem with memory capacity in the data entry section since only one month's data has to be held at any one time. A 15 by 15 product/territory matrix should be large enough for most applications. On the other hand, in the analysis section, it will almost certainly be necessary to compromise between the amount of data which can be handled and the variety of analysis options offered.

SALESTREND'S biggest memory problems occur in the sections which use graphics. With Mode 4 graphics, the 22k normally available for **BASIC** is reduced by some 12k leaving room for only a small amount of program and data. It is therefore essential to arrange the program so that the main body of data is held on a 'dump' file and the graphics program is called after the operator has decided on a limited number of displays.

The program's structure is outlined on the flow diagram overleaf:

SALES TREND – SIMPLIFIED BLOCK DIAGRAM



The main menu offers a choice between:

– Data entry (which enters the monthly sales data).

– Data analysis (which produces a display of a selected part of the database).

– Set up (used when the database is first set up to record the main parameters of the database).

– Delete (used to remove unwanted data files).

Set up:

The set up program maintains a record of the size of the data matrix (i.e. the number of products and territories) since is is essential that each monthly data file should contain the same number of data items.

Data entry:

The data entry program should be designed so that data can be entered relatively easily with a clear display and opportunities provided for editing mistakes both as they are made and as part of subsequent data checks. One approach is to enter just one column of data at a time (e.g. all the product sales in territory **1**) using a routine similar to that used in **GRAPHPLOTTER**. When all the data have been entered, the program should calculate totals and then display the complete table for final approval prior to saving it. It is useful to provide an opportunity for printing out the data so the operator has a permanent record on file.

The program for driving the printer will be quite long and should be arranged either as a subroutine or as a separate program called from the main menu. There needs to be some sort of 'scrolling' arrangement since 15 colums of data will probably not fit onto the printer. If the print program is intended to support different output formats and even different makes of printer, it will be necessary to input print constants (number of lines, single sheet/continuous stationery etc.) at the start of the program.

Data analysis:

To begin data analysis, the operator specifies which part of the database he wishes to examine. This is done by entering the following:

– Start and finish dates.

– The identity of the products and territories to be examined.

– Whether or not to adjust for inflation and working days.

– Whether a forecast is needed.

Entries must be carefully checked at this stage to ensure that they do not result in non-existent files being called or the memory capacity being exceeded. As memory usage is determined by a number of different operator-determined parameters, the formulation of the constraints needs close attention.

The program then reads the chosen files and extracts the data which relate to the selected products and territories, from each monthly file, putting it into memory in the form of a matrix of product/territory sales over time. If required, the data are adjusted for inflation and working days and again, if required, projected forwards using an exponential smoothing model like the one described in **ADJUSTER**.

The precise arrangement of data display will depend on the particular application of the program. Inevitably it is a compromise between a number of conflicting objectives. If the primary display is to be a graph, it is highly desirable to provide a means of easy reference to the numerical data on a secondary display. (No-one really believes a graph if they cannot see where it came from.) The graphical display itself has to be designed with some care. If all the product/territory variables are 'dumped' on the screen altogether, it will be impossible to distinguish one from another. If they are put up one after the other, the same problem will occur as the screen fills up. To get graphs up one at a time, each graph can be drawn and then deleted by redrawing it with the colour changed to black. Finally, to enable trends of different products and territories to be compared, the operator has to be able to select the graphs and specify the order in which they appear on the screen.

These multiple and conflicting requirements cause some nice programming problems. The displays have to be arranged so the operator can switch from one to the other using simple commands (preferably single keyboard entries) without either getting the graphs out of sequence or finding that the numerical data does not tie up with the display.

The basic scheme is to present the operator with display options in the form of a menu and then to put the whole display program into a loop. The operator starts by choosing between different alternatives on offer, then works through the chosen display before returning to the starting menu. There must also be an exit from the display back into the main program so that new data can be extracted from the database.

As with data entry, it is almost certain that hard copy of the analysis results will be required to provide a record of management decisions. Text output can be obtained by printing the data matrix using the same sort of program as was described under the data entry heading. For graphics output, there are three choices: photograph the screen, use a dot matrix printer or an X-Y plotter. Essentially, a dot matrix printer works off some sort of 'screen dump' in which the screen is scanned pixel-by-pixel. The printer makes either a dot or leaves a blank depending on what is present on the screen. An X-Y plotter is programmed in much the same way as graphs are drawn on the screen by defining the coordinates of the points which the pen is to join together.

In Conclusion

In the opening chapter "Computers — Friends or Foes", we drew attention to the disadvantageous position sales managers can be in if they do not have adequate factual information at their fingertips.

In the subsequent chapters our aim has been to present programs which enable the reader to collect factual data and present it in a managerial context.

This final chapter has been intended as an introduction to a more advanced type of programming, using the elements of the previous chapters.

The task of building up **SALESTREND** from it's constituent parts is to show how a sales manager can compete with accounting and production colleagues when it comes to explaining what is going on in the 'marketing mix'.

If, as is likely, you now have ideas for other programs not covered in this book, you will require some understanding of how the machine has been designed and programmed. This, in turn, implies a tentative step into computer technology beginning with the appropriate sections of your computer manual and moving onto specialist books and periodicals.

As it is always easier to get started by having a firm base to work from we hope that the programs supplied in this book will provide you with your foundation and that you will set to and start writing your own programs.

We hope that in a very short space of time you will be able to go to the next management meeting and say "My computer says . . . . ."

INDEX

INDEX

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

BUSINESS PROGRAMMING on your COMMODORE

This book is designed for
Sales and Marketing Managers
who have asked themselves the question
"How could I use our home computer
to help me with my day to day work
without having to spend a fortune on software?
Could I write my own programs
to handle sales forecasts,
customer record cards, graphs
and reduce my paper work load?"
The answer is *YES*
This book will show how it can be done.
Using basic
with notes for other types of machine
the user will be shown
how to program,
build own files systems using examples,
design a "Data Base"
which will form the core
for countless record systems.

The Author

Peter Jackson is a highly experienced businessman
who has served in senior management roles
with English Electric and Tube Investments.
He now has his own successful software company
and is a visiting lecturer
at the London Business school.

PHOENIX
PUBLISHING
ASSOCIATES

£7.95

ISBN 0-946576-19-X

90000

9 780946 576197