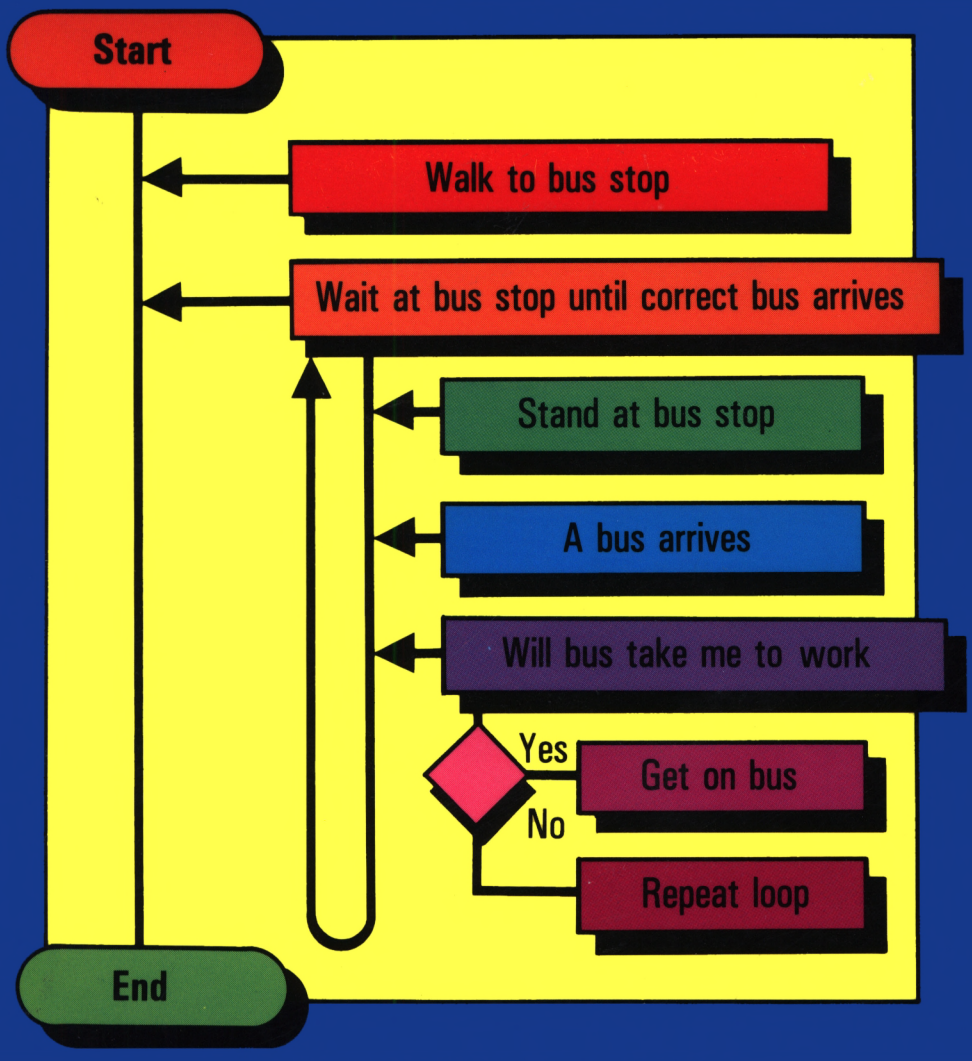
STRUCTURED PROGRAMMING

ON THE AMSTRAD COMPUTERS CPC 464,664 and 6128

Stephen Raven



MICRO PRESS

Structured Programming on the Amstrad Computers CPC 464, 664 and 6128

Structured Programming on the Amstrad Computers CPC 464, 664 and 6128

Stephen Raven

MICRO PRESS

First published in 1985 in the United Kingdom by
Micro Press
Castle House, 27 London Road
Tunbridge Wells, Kent

British Library Cataloguing in Publication Data

Raven, Steve
Structured programming on the Amstrad CPC 464, 664 and 6128.
1. Amstrad CPC 664 (Computer)—Programming
2. Amstrad CPC 464 (Computer)—Programming
3. Amstrad CPC 6128 (Computer)—Programming
I. Title
001.64'2 QA76.8.A4/

ISBN 0–7447–0034–5

Typeset by MC Typeset, Chatham, Kent
Printed by Mackays of Chatham Ltd

Contents

Section A Introduction to the Amstrad CPC 464, 664 and 6128 1

Chapter 1 The machine: its concept and breeding 3

Chapter 2 Commanding the CPC 664/464 by programming it ıing it with instructions 14

Section B Familiarity breeds confidence 29

Chapter 3 BASIC tools of the trade 39

Chapter 4 Debugging programs – editing facilities on the CPC 664/464 and how to use them 40

Section C The principles of BASIC 45

Chapter 5 Variable names and labels 47

Chapter 6 Reacting to your CPC 664/464 – `INPUT` statements 54

Section D From little blocks to structured programs 63

Chapter 7 Structured planning on CPC 664/464 applications 65

Chapter 8 Diagrams make the mind clearer 73

Chapter 9 Implementing the plan: 1 80

Chapter 10 Implementing the plan: 2 88

Section E Handling text – the key to information storage 101

Chapter 11 Strings, string variables and the $ symbol 103

Chapter 12 Manipulating strings to your advantage 108

Chapter 13 **READ** and **DATA** statements 116

Chapter 14 Extending variables – dimensional arrays 120

Chapter 15 Viewing the file, form fil, screen format, and data entry 124

Chapter 16 Processing numerical variables – automatic record updating 127

Chapter 17 Ideas for applications 132

Chapter 18 General principles and some reminders 139

Appendix 141

Index 149

Section A
Introduction to the Amstrad CPC 464, 664 and 6128

The CPC 6128

As this book was about to go to press, Amstrad launched the CPC 6128 home micro. Its advantages over the 664 are that it has twice as much RAM, a different keyboard (which isn't as good as the 664) and a new version of the CP/M business operating system which allows it to run a wider range of business programs.

The good news is that programs written in BASIC on the 464 and 664 will run without alteration on the 6128. This means that it should run all 664 disk based programs and 99.5% of 464 cassette programs. **It also means that you can use all of the programming hints and tips in this book to teach you how to use your new 6128.**

Although the 6128 is supplied with 128K of RAM, only 64K is available for BASIC programs. This is because the BASIC on the 6128 was originally designed for the 464 and 664 which only have 64K available. To help you get around this problem, Amstrad has included some extra BASIC commands which you can load from disk. These allow you to use the 'spare' 64K of RAM either as somewhere to store extra screen images or as a fast filing system. At the moment, you can't use the spare 64K to hold BASIC programs.

All these extra commands are installed by running the 'Bankmanager' machine code program supplied on your screen disk. If you want to use the spare RAM to hold extra screen images, Bankmanager supplies you with two extra commands:

'|SCREENCOPY' and '|SCREENSWAP' allow you to store up to four screen images in the extra RAM and then switch them onto the screen. This could be useful for games and animation, where you could set up new screens in the background and then switch them in when needed. Each screen image is given a 'Block' number from 1 to 5. To display an image you need to move it to Block 1.

|SCREENCOPY copies a whole screen image from a source Block to a target Block. The previous contents of the target Block are always overwritten and lost. For example if you want to save your display for later use, you would type something like '|SCREENCOPY,1,4'. This copies the image from Block 1 (which is always the display screen) to Block 4 (which is part of the spare RAM).

|SCREENSWAP just swaps the contents of two Blocks – it doesn't overwrite anything. For example |SCREENSWAP,1,4 would put the current screen image in Block 4 and the contents of Block 4 on the screen. Another |SCREENSWAP,1,4 would swap the images back to the way they started.

If you decide you want to use the spare 64K of RAM for storing data rather than screen images, the Bankmanager supplies you with four extra BASIC commands as follows:

|BANKOPEN allows you to specify how many characters will be in each record. The maximum length is 255 characters. |BANKWRITE lets you write a record, |BANKREAD lets you read a record and |BANKFIND does a string search to find a matching record and returns the record number so that you can then do a |BANKREAD to recover the data.

All the commands apart from |BANKOPEN need you to specify an integer variable to hold the status code and a string variable to hold the data to be read or written to the file.

Chapter One

The machine: its concept and breeding

The computer is a very complex machine, able to respond in a precise and an accurate manner to a series of instructions.

A series of instructions is placed within the computer by means of a computer program. A program is defined as a list of instructions that will cause the computer to do a clearly defined job of work. This book is primarily concerned with how to create programs for the Amstrad CPC 664 and the CPC 464 with a style and structure that will ensure your hopes and objectives for your programming projects are fulfilled. Such general principles are applicable to all computers but the Amstrad CPC 664 and CPC 464 as machines have several advantages for the purpose of creating well structured programs with the aim of being useful about the home.

The advantages are:

1) Both computers allow the user to communicate effectively in a form of BASIC—the language we give instructions to the computer by—that is easily read with a minimum of code numbers and meaningless formulae.
2) The documentation with the computers is comprehensive in its description of the commands available to the user to communicate with the machine. Using those commands together with this book, the extent of programming ability in the BASIC language will only be limited by the programmer's imagination.
3) The equipment supplied with the CPC 464, i.e. the cassette recorder and monitor, means an economic purchase, ensuring further demands on the household's TV set will not be made. There are distinct disadvantages in using a TV screen if you require to look up how to make Granny's

Christmas Cake on your recipe database if someone else is watching their favourite TV soap opera at that particular moment.

4) The CPC 664 has a very distinctive advantage in terms of having disk storage alongside the keyboard. This package means not only an economic purchase, but also a time saving element in loading and storing programs associated with a disk system as compared to a cassette storage system.

In order to give instructions to a computer they must be presented in a very precise form, so that the computer can understand and hence execute them. One of the purposes of the keyboard is to deliver these instructions to the advanced technology held inside the casing of the computer, rather like giving instructions to another person, and then that person carrying them out. This person would interpret the instructions and even fill in the bits he or she did not hear or understand. Just like any computer the Amstrad CPC 664/464 cannot interpret instructions outside of its language format. The instructions have to be delivered in a form, which is known as the *syntax*, of the CPC 664/464 BASIC language which obeys the rules written into the machine's specification.

To become a proficient CPC 664/464 user, developing useful program applications with a style of program structure that will guarantee successful implementation of your ideas, there are two skills you must aspire to. The first is to be able to communicate with the CPC 664/464 in *its* language, which is a little like yours but more logical and definitely more precise in its presentation. If you deliver an instruction that the CPC 664/464 does not understand it will respond with one of a number of messages known as 'error messages' which will provide you with a clue of why the CPC 664/464 cannot understand your instruction and therefore cannot carry it out. It is the informed use of these error messages that make up one of the essential elements of a good CPC 664/464 program creator.

The second skill you have to aspire to is the ability to develop your thoughts in such a way that the application is planned in minute detail, the whole structure is designed in the form of blocks that when placed together will perform the

required task accurately and repeatedly. During this planning stage pen and paper and the use of diagrams are the essential tools. This is not to say do not use the CPC 664/464 to experiment your ideas on, but always after trying the idea return to planning the whole project before keying the finished program into the computer.

As you type in your instructions, it would be very unusual if you did not make some typing errors. The CPC 664/464 will respond with 'error messages'. In order to learn quickly and effectively, it will be necessary to swallow your pride and accept that you have to keep within the limitations of the CPC 664/464's language. Use the 'error messages' Appendix VIII in the CPC 464's *User Instruction Manual*; CPC 664 users should refer to Chapter 7 Part 6 of their manual. Then correct the mistakes in the instructions. Only then will you be able to command the CPC 664/464 to follow your wishes.

A computer is patient since it will wait for you to make a move; it will repeat the same response for ever if you do not make any changes to the instructions held in its memory. If such instructions include parts which the computer cannot interpret, an 'error message' will be displayed on the screen. The computer cannot miraculously correct the parts it cannot interpret. It is essential to bear in mind that the computer will not, or more precisely cannot, change its dialect of logical and precise commands to your dialect which is, though I hate to say it, typically human, containing numerous elements open to various interpretations by the person listening to you.

A computer responds to lists of instructions, which are known as programs; they are typified by having a logical order, as each line of instructions begins with a line number, ascending in magnitude as the program progresses. In order to develop new skills we need to concentrate on the CPC 664/464's ability to respond to a single line of instructions immediately: this is known as direct command mode. There are several uses for this facility while actually creating a program; it may also be used as an application even before you have produced a program.

Set up your computer and switch the power on so that you have the manufacturer's initialisation message on the screen. Type in the letters `CLS` or `cls` and press the <**ENTER**> key (a blue key on the keyboard). The screen should now be

cleared of the manufacturer's information, with only the word **READY** and a yellow square (the text cursor) in the top left hand corner. This is a very simple direct command that you have caused the CPC 664/464 to execute. Note the CPC 664/464 allows you to enter its instructions in both upper or lower case letters, an excellent feature of these machines.

From now on specific keys to be pressed will be placed within these symbols <> e.g. <**ENTER**> means to press the **ENTER** key.

The benefits of using direct command mode are:

1) Use of the direct command mode allows the machine user to gain familiarity with the facilities available and insight into their potential. (Try some BASIC commands out now. Look at the BASIC keyword reference of your machine's *User Manual* and remember to take note of the *syntax* requirement, e.g. try PEN 4 <**ENTER**> to start with.)
2) Direct command mode develops an understanding of the machine's logical workings and the speed of working, for example, to do arithmetical calculations.
3) As the user's programming abilities develop, direct command mode enables the use of the CPC 664/464 as a jotting pad, scribbling down ideas to try them out before entering them into a program.
4) Use of the direct command mode will illustrate the frustration of not being able to store the instructions for long term use, as with a computer program.

The home micro's primaeval grandfather is the electronic calculator. Just as we would expect with all technological advances, more and more facilities are now available. The home micro is no exception. Rather than merely having a number of LED displays as a calculator does, displaying only numbers, there are several output facilities which can respond by sending all kinds of numbers, characters, and other symbols to the monitor screen, to a cassette in the cassette recorder and, if fitted, to a printer to provide paper copy of the output. For these reasons the commands, for example of how the answer to a calculation is to be displayed, have to be explicit. The instructions to perform a calculation have to include not only the sum itself, but in what form and to which device—monitor, printer or cassette—where—the location on

the screen, etc.—and when—at what time and for how long—the answer is to be displayed.

Now type in the instructions, exactly as they appear in Figure 1.1 and remember to press the <**ENTER**> key once you have finished each line. To print the + sign hold the <**SHIFT**> key down and press the appropriate key.

```
PRINT 6+5                <ENTER>
      └──────── This is a blank space, which must be included.

PRINT TAB(15) 23+4       <ENTER>
      └──────── This is a blank space, which must be included.
```

Figure 1.1 Direct command mode arithmetical calculations

The answers to the sum will appear on the screen, one line below the instruction line.

Now experiment with the number inside the brackets. How long is a line? What is the largest number you can place inside these brackets? Try placing a sum (two numbers to be calculated) inside the brackets. Do not forget to retype the rest of the instruction exactly as in the original instruction.

You have now probably discovered that the **TAB** command allows up to 80 characters or two lines of positioning on the screen. Further arithmetical formulae can be used to position the answer on the screen.

What type of message appears on the screen if the blank spaces are omitted? If you had omitted the blank space after the **PRINT** command one of two error messages would have appeared on the screen. With only the **PRINT** statement used the message would be 'Syntax error': the CPC 664/464 cannot interpret your instructions as governed by the CPC 664/464 language rules. If the **TAB** command is also included, the error message will depend on the number placed inside the brackets, determining how the computer interprets your instructions. If the number is less than 11 a 'Syntax error' message will be printed on the screen. If the number is 11 or greater 'Subscript out of range' will be printed on the screen; this will be explained in greater detail later, but effectively the CPC 664/464 is being confused in what it is being asked to do.

However, most syntax errors are the result of typing

mistakes or the omitting of a blank space which is essential for clarity of the instruction, i.e. the blank space has the effect of terminating a CPC 664/464 BASIC command word.

The symbols the CPC 664/464 uses to do arithmetical calculations are not the ones you are probably familiar with. They are different only for reasons of legibility and therefore will be more easily recognisable. A summary of important symbols and arithmetical operators is shown in Figure 1.2.

Symbol	Arithmetical and Relational Operators
/	division
*	multiplication
+	addition
-	subtraction
\	Integer division the result will always be a whole number.
MOD	Gives the remainder when an integer division is carried out.
↑	Exponentiation, raises the number to a power, ie. 4 2 is the square of four, answer 16
=	is equal to
<	is less than
>	is greater than
<=	is less than or equal to
=> or >=	is greater than or equal to
<>	is not equal to

Figure 1.2 The CPC 664/464's arithmetical and relational symbols

All of the symbols and combinations in Figure 1.2, appear on the keyboard. If it is the lower symbol on the key you can enter it directly. If it is the upper symbol on the key you will have to press and hold the <**SHIFT**> key down, while you press the appropriate symbol key.

Experiment by doing further calculations of a more complex nature. The CPC 664/464 performs its arithmetic in specific order of calculation: all multiplication elements are performed first, followed by the division, addition, and then subtraction elements. As in traditional mathematics the use of brackets is

permissible and signifies that this part of the calculation is to be performed before the rest of the sum is calculated.

Hand written expression	CPC464's expression
$\frac{(6+5)}{(6\times(5+6))}$	PRINT (6+5)/(6*(5+6)) 0.166666667
$\frac{11}{6\times11} = \frac{1}{6}$	

Figure 1.3 Comparison of handwritten and CPC 664/464's arithmetical expressions

Punctuation for the **PRINT** statement is important. Try this:

```
PRINT 3+2,3-2,3*2        <ENTER>
PRINT 3+2;3-2;3*2        <ENTER>
```

Figure 1.4 Punctuation and the **PRINT** statement

Commas(,) when combined with the **PRINT** command allow the output of information to be displayed in one of three columns across the screen. The columns are filled consecutively from the left hand side; when all three columns are filled the cursor automatically moves on to the next line.

Semi-colons(;) used while printing numbers will cause the number to be printed with a blank space on either side of it.

If you happen to enter a letter instead of a number, you will see on pressing the <**ENTER**> key a zero; you have created a variable that has the value of zero, which is explained in Chapter 5, 'Variables and the **LET** statement'. For the time being try typing this into the CPC 664/464:

```
number=12:PRINT number  <ENTER>
```

A colon(:) allows the CPC 664/464 user to combine a variety of commands and statements on one line of instructions. A line of instructions can consist of up to 255 characters, not merely a line across the monitor screen.

Now let us experiment by typing the following commands into your CPC 664/464, pressing the <**ENTER**> key, when-

ever instructed to do so:

a) `MODE 0` <**ENTER**>
b) `MODE 1` <**ENTER**>
c) `MODE 2` <**ENTER**>

Press and hold down <**SHIFT**> and <**CTRL**>, while pressing <**ESC**>.

d) `PEN 4:PAPER 2:BORDER 3` <**ENTER**>
e) `CLS` <**ENTER**>

Repeat instructions (*d*) and (*e*) using different numbers. Find any limitations there may be which produce an error message. Try changing the `MODE` by typing either line (*a*) or (*c*), again experimenting with a range of numbers.

To reset the CPC 664/464 hold <**SHIFT**> and <**CTRL**> and press <**ESC**>.

Extending the instruction line by the use of colons(:) can be advantageous both for programming reasons and also for the purpose of trying ideas out, in direct command mode, very similar to using a note book or jotting pad.

```
FOR counter=1 TO 10:PRINT "CPC464":NEXT
```

It would now appear that there are no limitations to the extent of the instructions that can be given to the CPC 664/464. Unfortunately this is not so; due to the construction of most computers there is a limit to the number of characters that can be used in any one instruction line. In the case of the CPC 664/464 the maximum is 255 characters. A long term disadvantage with direct commands is the inability to store them in the computer's memory for repeated use. Consequently the instructions have to be retyped every time the response is required. By adding a line number the execution of the instructions by the CPC 664/464 can be repeated as many times as the user requires them to be executed, simply by typing the command word `RUN` and hitting the <**ENTER**> key.

```
10 FOR counter=1 TO 10:PRINT "CPC464":NEXT
RUN <ENTER>
```

Applications in direct command mode

The most obvious use for the CPC 664/464 in direct command mode is that of a calculator, with the ability to calculate complicated formulae with one swift press of a key. Follow through the example in Figure 1.5.

```
PRINT "Amount invested = £25.00": LET principal=25      <ENTER>
PRINT "Rate of interest = 12%": LET rate=12             <ENTER>
PRINT "Time (in years) = 2yrs": LET time=2              <ENTER>

LET interest=(principal+rate*time)/100                  <ENTER>
LET balance=principal+interest                          <ENTER>

PRINT "The interest payable on this investment is: £";interest <ENTER>
PRINT "The new balance is.....: £";balance              <ENTER>
```

Figure 1.5 Calculator application in direct command mode

This demonstrates a useful and informative means of doing a series of calculations. By examining this application, it should be possible for you to see the principle that commands such as **PRINT** only send information to the screen. It is therefore also necessary to instruct the computer memory with the same information as you are reading on the screen; this is done by the use of such commands as the **LET** statement.

Let us now consider a different approach to the use of direct command mode. Reset your CPC 664/464 by holding down <**SHIFT**>, <**CTRL**> and pressing <**ESC**>.

By following the instructions in Figure 1.6 it will be possible for you to design room layouts etc., using a range of the graphics facilities available on the CPC 664/464.

```
MODE 0                                                  <ENTER>
WINDOW 1,20,4,4 : PAPER 3 : PEN 2 : CLS                 <ENTER>
PRINT "1=Yel 2=Blue 0=Erase"; : WINDOW 1,20,1,3         <ENTER>
PEN 3 : PAPER 1 : CLS                                   <ENTER>
ORIGIN 20,20,20,620,320,20                              <ENTER>
DRAW 600,0,2 : DRAW 600,300 : DRAW 0,300 : DRAW 0,0     <ENTER>
```

Figure 1.6 Instructions in direct command mode

Now experiment for yourself. If you get into too much of a mess, reset the CPC 664/464 and start from the beginning of the instructions in Figure 1.6.

To draw in lines use either of the draw statements. **DRAW** uses absolute coordinates, where the position 0,0 is the bottom left hand corner. The area available for your room plan designs etc., has a maximum x coordinate of 600 and a y coordinate of 300.

```
Command         Syntax
MOVE            MOVE x coordinate, y coordinate
MOVER           MOVER x value, y value                  cursor moves relative to
                                                        previous cursor position
DRAW            DRAW x coordinate, y coordinate, color
DRAW            DRAWR x value, y value , color          cursor moves relative to
                                                        previous cursor position
      color = a number 1=yellow lines etc.
              The colour will remain until it is subsequently
              changed, by a following DRAW or DRAWR statment.
              Erase by using color=0 ie. DRAWR 100,100,0 will
              draw a line the same colour as the paper/background
              diagonally 100 units up and 100 units across.

    The blue rectangle represents a drawing area, which has an maximum x
coordinate of 600 and a y coordinate of 300. The MOVE and DRAW statments positon
the cursor with respect to this grid. The MOVER and DRAWR statments are the most
useful, in that you can mentally split the rectangle into a grid of 600 across,
300 up, add or subtract the appropriate quantity to draw a line or move the
cursor.
```

Figure 1.7 Instructions for using the direct command mode design screen

The **DRAWR** statement uses coordinates relative to the last position of the graphics cursor, i.e. to draw a vertical line upwards from the current position, use in the y coordinate's position in the statement a value equivalent to the length of the line required, while the value in the x coordinate's position will be zero. To draw a vertical line downwards from the current cursor position repeat the process, but include a minus sign as a prefix to the y coordinate value.

To erase a line, repeat the previous set of commands causing the unwanted line(s), having first moved the cursor to the start of the section of unwanted line(s), and ending your **DRAW** or

DRAWR command with a zero.

Remember, if you have an idea, try it out. Experiment, see if you can improve on this application. Once you have worked through the rest of this book, return to this application and see if you can convert it from direct command mode into a complete program and therefore save it on cassette tape, ready for repeated use.

Chapter Two

Commanding the CPC 664/464 by programming it with instructions

A computer application will consist of a series of instructions held in its memory in order to carry out a very specific task. These instructions will therefore be required on numerous occasions, making it totally impractical to type them into the CPC 664/464 every time they are required. Long term storage of programs is an extremely valuable facility and when using the CPC 464 the best method is to use the data recorder to the right hand of the keyboard. The CPC 664 has a distinct advantage because of the presence of a disk storage system, a system which will enable very effective data file management to be arranged. The example from the previous chapter would not fit these requirements, having to be set up each time it is to be used.

The application we are now going to consider is in fact a suite of three programs that when used in conjunction with each other can provide an effective and tidy method of storing up to one hundred names and telephone numbers in the form of a computer data file. Once this information has been created and stored on data cassette, it will be possible to add further names and numbers at a later date—up to the maximum allowed—browse through your personalised telephone directory at the press of a key, select a first name or surname and view all entries in your directory under that particular name.

Before we actually start the work of putting this program into your CPC 664/464, you should bear in mind the following points:

1) The following program is a foundation, to develop and add to as your programming ability spreads its wings. Throughout each chapter hints and ideas will be made to further improve the facilities available to the program user.
2) The design and structure built into these programs was initiated on paper before sitting down at the CPC 664/464 to key it in. If that approach was followed in this type of book you would be reading about 'something' being created while having no idea of the finished article. You need to become acquainted with the product and be effective in its use before you can work through the why's and how's of its creation and application.

Turn on your computer and ensure that there is nothing stored in its memory by using the <**SHIFT**>, <**CTRL**> and <**ESC**> keys. When entering a program remember to press the <**ENTER**> key at the end of each program line. Before you start it is possible to put your computer into automatic line numbering mode. The following command will start numbering lines at line 10 and every time the <**ENTER**> key is pressed will increase the line number by 10.

`AUTO line number, increment` <**ENTER**>
e.g. `AUTO 10,10` <**ENTER**>

To leave automatic line numbering mode press the <**ESC**> key. To return to automatic line numbering use the `AUTO` command, but change the first number of the command to the next line from the program listing that you require to enter into the CPC 664/464's memory.

Part one: The option menu

The aim of this program is to be able to choose between creating/adding to the telephone directory or using the facilities available to look through the information stored in the directory.

```
10 REM Personalised Telephone Directory
20 REM Steve Raven  March 1985
30 REM Part one  Option Menu
```

```
40 CLS:PEN 3
50 FOR display=1 TO 22
60 PRINT TAB (display) "Telephone Directory"
70 NEXT
80 WINDOW 2,40,2,11
90 CLS:PRINT:PEN 2
100 PRINT"Do you require to:"
110 PRINT:PRINT"<A> Create or add to your directory "
120 PRINT:PRINT"<B> Use your directory"
130 PRINT
140 PRINT "Press the appropriate <key>"
150 WINDOW 2,40,13,21
160 CLS:PRINT:PRINT
170 PRINT "Remember you must have created a "
180 PRINT
190 PRINT "directory before you can use it."
200 PRINT
210 WHILE K$<>"A" AND K$<>"B"
220 K$=INKEY$
230 K$=UPPER$(K$)
240 PEN 1
250 WEND
260 IF K$="A" THEN RUN"Create"
270 IF K$="B" THEN RUN"Use"
280 END
```

Now carefully check what you have typed in, a few lines at a time, by using the LIST command:

LIST 10–200 <**ENTER**>

This command will actually list all the lines you have typed in *and* entered from line number 10 up to and including line number 200. If a complete line is missing you merely have to type it in and press <**ENTER**>. The next time you use the LIST command it will appear in the listing. If you find a typing mistake, for the time being retype the whole line again and press the <**ENTER**> key. Later on we shall examine much more effective methods of editing program lines, using the facilities available on the CPC 664/464. Once all these lines have been checked move on to check lines 210 to 280 by using LIST 210–280 <**ENTER**>. Once you are satisfied that the program is entered into memory correctly, type in the command RUN and press the <**ENTER**> key.

Remember you cannot actually expect the program to fulfil its aim of loading one of the two programs it asks the user to select, as you have not yet created them or stored them on the particular form of storage you have with your computer, so use this method to check that you have made no typing mistakes, i.e. no 'error messages' will appear on the screen after the command `RUN` has been entered if the program has been entered exactly as in lines 10 to 280 above.

The program should execute its instructions, asking the program user to select one of two options. You will find that you will only be able to press the appropriate keys or the <**ESC**> key to have any effect. This is known as validating the program user's response to the questions asked, i.e. if this was not the case the program would not do its stated task of either creating or using the telephone directory. The reason we have still maintained the use of the <**ESC**> key is so that we can return to the program listing. If an 'error message' is encountered type `MODE 1:LIST` <**ENTER**> and recheck the line that the error was reported on. Note that blank spaces are important and retype the line as it is in the program listing. If no 'error messages' appear then save the program on your particular storage system. For CPC 464 system users follow the instructions in the first list below. CPC 664 users should skip the first and follow the second one. For further details consult the CPC 464 user's manual. If after returning to the program listing you want to view the instructions in the original colour, type `PEN 1` <**ENTER**>. If you do not change the `MODE` before using the command `LIST`, the program listing will appear in one of the 'windows' set up by the program itself.

CPC 464 users:

1) Place the blank cassette in the data recorder, reset the tape counter to zero, fast forward the tape so that the counter reads 025.
2) Type the command `SAVE "PHONEHOME"`, press <**ENTER**> key.
3) Follow the instructions which appear on the screen.
4) Verify the program is saved on the cassette by rewinding the cassette so that the tape counter reads 023, type the command `CAT`, press **PLAY** on the recorder and the <**ENTER**> key. The relevant information will appear on

the screen as the option menu program is only one block in length. The screen will read: `PHONEHOME block 1 $ ok` as long as the program is saved on the cassette.

5) If the response to the `CAT` command is not as above, repeat the saving process.

CPC 664 users:

1) Place a new previously `FORMATTED` disk into the floppy disk drive (consult the machine's manual on how to `FORMAT` a disk).
2) Type the command `SAVE "PHONEHOME"` press <**ENTER**> key.
3) Follow the instructions which appear on the screen.
4) Verify the program is saved on the disk by typing the command `CAT` and pressing the <**ENTER**> key. The screen will display all the files stored on that particular disk.
5) If the list does not include the name `PHONEHOME`, repeat the saving process until the required `CAT` response is obtained.

Part two: Creating or adding to the directory

Part two of the telephone directory program creates the directory or adds names and numbers to a previously created directory.

Lines 10 to 190 consist of the program's initialisation and main control element. These instruction lines can be considered as the foundation to the rest of the program, ensuring the subroutines are executed in the right sequence and at the appropriate time.

```
10 REM Create a telephone directory
20 REM or add new names and numbers
30 REM to a previously created
40 REM directory
50 REM Part two     Create
60 :
70 t=100:counter=0:record=0
80 DIM name$(t),surname$(t),phone$(t)
90 f$="Firstname":s$="Surname":ph$="Phone No."
```

```
100 MODE 1: PEN 1
110 :
120 GOSUB 210       :REM Option Menu
130 IF k$="A" THEN GOSUB 310
140                 :REM add to directory
150 IF k$="C" THEN GOSUB 620
160                 :REM create directory
170 GOSUB 960       :REM save directory
180 GOSUB 1190      :REM use or finish
190 END
```

Program lines 210 to 300 relate to the subroutine which will provide the program user with the information for him or her to make a response.

```
210 REM Option menu on screen
220 LOCATE 1,2:PRINT"Choose the facility you require:"
230 LOCATE 5,4:PRINT"<C>reate a NEW directory"
240 LOCATE 5,6:PRINT"<A>dd to an OLD directory"
250 LOCATE 1,8:PRINT"Press either key <C> or <A>"
260 WHILE k$<>"A" AND k$<>"C"
270 k$=INKEY$
280 k$=UPPER$(k$)
290 WEND
300 RETURN
```

The following program lines contain the instructions which enable the computer to perform the first of the two facilities available to the program user of this part of the application. The **REM** statement line explains what it does.

```
310 REM Add new phone nos to directory
320 CLS
330 LOCATE 5,24:PEN 5
340 PRINT"Insert data cassette in datacorder"
350 LOCATE 1,1
360 FOR delay=1 TO 1500:NEXT
370 CLS
380 OPENIN"data"
390 WHILE EOF=0
400 INPUT#9,name$(counter)
410 INPUT#9,surname$(counter)
420 INPUT#9,phone$(counter)
430 PRINT counter,
```

```
440 counter=counter+1
450 WEND
460 CLOSEIN
470 record=counter
480 WINDOW 1,40,1,23:CLS
490 PRINT f$,s$,ph$
500 counter=0
510 WHILE counter<record
520 PRINT name$(counter),surname$(counter),phone$(counter
530 counter=counter+1
540 WEND
550 WINDOW 1,40,24,24
560 PRINT"Press key <c> to continue"
570 WHILE k$<>"C"
580 k$=INKEY$
590 k$=UPPER$(k$)
600 WEND
610 RETURN
```

CPC 664 USERS:

```
340 PRINT "Insert disk into floppy disk drive"
```

Lines 620 to 950 provide the program for the second facility available in this part of the application.

```
620 REM create directory
630 WINDOW 1,40,1,25: CLS
640 PRINT SPC(10);PRINT"Information Entry";PRINT
650 PRINT f$,s$,ph$
660 WHILE counter<100
670 record=counter+1
680 WINDOW 1,40,11,11
690 IF k$=CHR$(32) THEN PEN 2
700 IF k$<>CHR$(32) THEN PEN 5
710 k$=""
720 PRINT"Record Number: ";record
730 WINDOW 1,40,5,9
740 PRINT SPACE$(80)
750 LOCATE 1,2
760 LINE INPUT; name$(counter)
770 LOCATE 14,2
780 LINE INPUT; surname$(counter)
790 LOCATE 27,2
800 LINE INPUT; phone$(counter)
 10 WINDOW 1,40,17,25
```

```
820 PRINT"Press the <ENTER> key if data is correct"
830 PRINT:PRINT"If not press the <SPACE> bar"
840 PRINT:PRINT"If END of DIRECTORY press the <E> key"
850 WHILE k$<>CHR$(13) AND k$<>CHR$(69) AND k$<>CHR$(32)
860 k$=INKEY$:k$=UPPER$(k$)
870 WEND
880 CLS
890 IF k$=CHR$(13) GOTO 930
900 IF k$=CHR$(69) GOTO 920
910 IF k$=CHR$(32) GOTO 680
920 counter=99
930 counter=counter+1
940 WEND
950 RETURN
```

Once the program user has added further names and numbers or created a new directory, clearly the information has to be stored in some form for use in conjunction with part three of the application. The program lines 960 to 1180 perform this function.

```
960 REM save the created directory
970 REM on data cassette
980 PEN 5
990 WINDOW 1,40,1,25: CLS
1000 WINDOW 1,40,11,11
1010 PRINT SPC(5);"Total number of records: "record
1020 WINDOW 11,30,20,24
1030 PEN 7: CLS
1040 PRINT SPC(1)"Save Directory"
1050 PRINT:PRINT"on data cassette"
1060 WINDOW 1,40,1,8
1070 PEN 1
1080 LOCATE 1,1
1090 OPENOUT "data"
1100 counter=0
1110 WHILE counter<record
1120 PRINT#9,name$(counter)
1130 PRINT#9,surname$(counter)
1140 PRINT#9,phone$(counter)
1150 counter=counter+1
1160 WEND
1170 CLOSEOUT
1180 RETURN
```

CPC 664 USERS:

```
970 REM on floppy disk drive
1050 PRINT:PRINT"on floppy disk drive"
```

Lines 1190 to 1300 merely enable the program user to finish using the program or to move on to part three of the application and subsequently use that.

```
1190 REM Use directory or finish
1200 WINDOW 1,40,1,25: CLS
1210 PRINT:PRINT"Press the appropriate <key>"
1220 LOCATE 5,5: PRINT"<Q>uit"
1230 LOCATE 5,10: PRINT"<U>se the directory"
1240 WHILE k$<>"Q" AND k$<>"U"
1250 k$=INKEY$
1260 k$=UPPER$(k$)
1270 WEND
1280 IF k$="Q" THEN CLS
1290 IF k$="U" THEN CLS: RUN"Use"
1300 RETURN
```

Repeat the process of checking the program lines for typing mistakes by using the **LIST** command and retyping the offending lines as appropriate.

If you want to check that the CPC 664/464 will execute the program use the **RUN** command. To return to the program listing always use this method: Press the <**ESC**> key, type **MODE 1:LIST** and press the <**ENTER**> key.

CPC 464 users: Once you are satisfied the program is as presented on these pages position another blank cassette in the data recorder and **SAVE** the program under the name 'Create' i.e. **SAVE "Create"** <**ENTER**> and verify it has been saved correctly.

CPC 664 users: Once you are satisfied the program is as presented on these pages place the disk in the floppy disk drive and **SAVE** the program under the name 'Create' i.e. **SAVE "Create"** <**ENTER**> and once the operation is complete verify it has been saved by the use of the **CAT** command.

Part three: Using the telephone directory

The first section of the program, as in part two, is the initialisation and main control element of the program.

```
10 REM Use Telephone Directory
20 REM Part three   Use
30 :
40 t=100
50 DIM name$(t),surname$(t),phone$(t)
60 counter=0
70 f$="Firstname":s$="Surname":ph$="Phone No."
80 MODE 1:PEN 1
90 :
100 GOSUB 200     :REM load data file
110 WHILE k<>9
120 GOSUB 380    :REM select facility
130 ON k GOSUB 550,710,950,1230
140 WEND
150 REM 550-browse through directory
160 REM 710-select and search
170 REM 950-amend a record
180 REM 1230-quit
190 END
```

Lines 200 to 370 present the routine of instructions that load into the CPC 664/464 the data previously stored on cassette or disk by the program user, i.e. the names and numbers in the telephone directory.

```
200 REM load data file
210 CLS
220 LOCATE 10,10:PRINT"Use Your Directory"
230 LOCATE 5,24
240 PRINT"Insert data cassette in datacorder"
250 LOCATE 1,1
260 FOR delay=1 TO 2500:NEXT
270 LOCATE 1,1: PEN 2
280 OPENIN"data"
290 WHILE EOF=0
300 INPUT#9,name$(counter),surname$(counter),phone$(counter)
310 PRINT counter,
320 counter=counter+1
330 WEND
```

```
340 CLOSEIN
350 record=counter
360 OPEN 1
370 RETURN
```

CPC 664 USERS:

```
240 PRINT "Insert disk in floppy disk drive"
```

Lines 380 to 540 present on the screen the information the program user will require to make an appropriate response.

```
380 REM select facilties
390 WINDOW 1,40,1,25: CLS
400 PRINT TAB(8)"Facilities Available"
410 LOCATE 5,3
420 PRINT"Choose each option by pressing the"
430 PRINT"appropriate number and hit the <ENTER>  key"
440 LOCATE 8,7
450 PRINT"<1> Browse through directory"
460 LOCATE 8,9
470 PRINT"<2> Select and search"
480 LOCATE 8,11
490 PRINT"<3> Amend a record
500 LOCATE 8,13
510 PRINT"<4> Quit
520 LOCATE 8,15
530 INPUT k
540 RETURN
```

The remaining subroutines perform the facilities available to the program user. The specific functions are indicated by the **REM** statement on the first instruction line of each subroutine.

```
550 REM browse through directory
560 CLS
570 PRINT f$,s$,ph$
580 WINDOW 1,40,3,18
590 c=0
600 WHILE c<record
610 PRINT name$(c),surname$(c),phone$(c)
620 page=c MOD 14
630 IF page=0 AND c>0 THEN GOSUB 1280
640 IF page=0 AND c>0 THEN WINDOW 1,40,3,18: CLS
650 c=c+1
660 WEND
```

```
670 WINDOW 8,32,20,23
680 PRINT"End of Directory"
690 GOSUB 1280
700 RETURN
710 REM select and search
730 CLS
740 PRINT"Do you want to select by: "
750 PRINT:PRINT"<A> First name
760 PRINT:PRINT"<B> Surname"
770 WHILE k$<>"A" AND k$<>"B"
780 k$=INKEY$
790 k$=UPPER$(k$)
800 WEND
810 PRINT
815 name$="zz":surname$="zz":f=0
820 IF k$="A" THEN INPUT"Which first name";name$
830 IF k$="B" THEN INPUT"Which surname";surname$
840 PRINT:PRINT "   ";f$,s$,ph$
850 WINDOW 1,40,10,21
860 c=0
870 WHILE c<record
880 IF k$="B" THEN 900
890 IF name$=name$(c) THEN f=1: PRINT c;name$,surname$(c),phone$(c)
900 IF surname$=surname$(c) THEN f=1: PRINT c;name$(c),surname$,phone$(c)
910 c=c+1
920 WEND
925 IF f<>1 THEN LOCATE 10,8:PRINT " No Record Found "
930 GOSUB 1280
940 RETURN
950 REM amend a record
960 CLS
970 PRINT"Please select record for amending by: "
980 GOSUB 750
985 IF f<>1 THEN RETURN
990 WINDOW 1,40,22,24
1000 INPUT"Input the number of the record you want to change";n
1010 CLS: PRINT"Re type the whole record please."
1020 WINDOW 1,40,10,21: CLS
1030 PRINT n;name$(n),surname$(n),phone$(n)
1040 LOCATE 2,4: INPUT;name$(n)
1050 LOCATE 12,4: INPUT;surname$(n)
1060 LOCATE 25,4: INPUT;phone$(n)
1070 WINDOW 1,40,1,25
1080 CLS
```

```
1090 PEN 7
1100 PRINT"Save Directory on Data Cassette"
1110 LOCATE 1,22
1120 OPENOUT"data"
1130 c=0
1140 WHILE c<record
1150 PRINT#9,name$(c)
1160 PRINT#9,surname$(c)
1170 PRINT#9,phone$(c)
1180 c=c+1
1190 WEND
1200 CLOSEOUT
1210 PEN 1: PAPER 4
1220 RETURN
```

CPC 664 USERS:

```
1100 PRINT "Save Directory on Disk"
1230 REM quit
1240 CLS
1250 LOCATE 10,10
1260 PRINT"That's All Folks!"
1270 RETURN
1280 REM press any key routine
1290 WINDOW 1,40,25,25
1300 PAPER 1: PEN 3
1310 CLS
1320 PRINT TAB(8)"Press any key to continue"
1330 k$=INKEY$: IF k$="" THEN 1330
1340 PAPER 4: PEN 1: CLS
1350 WINDOW 1,40,1,25
1360 RETURN
```

CPC 464 users only: Once you are satisfied the program you have typed into your computer is the same as that which appears on these pages save it on a third cassette using **SAVE "Use" <ENTER>**, etc. and again verify that it has been saved correctly.

You should by now have three cassettes with the three parts of the application stored separately, one program on each cassette. Clearly the application is quite usable in this form as long as the user is prepared to put in the appropriate cassette when the CPC 464 is attempting to load that specific program. In addition the user must not forget to place a blank cassette in the recorder when loading or saving the name and telephone

number information. For the purpose of convenience, it could be worth while loading part one into the CPC 464 and saving it on another blank cassette, loading part two and saving it consecutively on the same blank cassette, and repeating the process for part three. The result should be a second copy of each program, but stored on the same cassette, which saves having to keep changing the cassettes round. Note that the names and telephone numbers data should, however, be stored on a separate cassette.

If you were to use the **CAT** command with the program tape in the recorder, the screen should display the information as shown in Figure 2.1. The string **($)** symbol illustrates that the information stored on the cassette is of the program instruction type, rather than data type as in the names and numbers stored on the other cassette; a * symbol illustrates data is stored on the cassette.

```
Ready
CAT
Press PLAY then any key :
PHONEHOME              block 1 $ ok
Create                 block 1 $ ok
Create                 block 2 $ ok
Use                    block 1 $ ok
Use                    block 2 $ ok
```

Figure 2.1 Screen display if using **CAT** command on program tape

CPC 664 users: Save this program on the same disk as previously used and in the same manner; this time the file should be labelled '**Use**'. Once more verify it by using the command **CAT**. All three programs should now be stored on the one disk.

Reset your computer in the normal way as explained previously. Type the command **RUN** followed by the file name **PHONEHOME** inside inverted commas thus: **RUN-"PHONEHOME" <ENTER>**.

Telephone directory application operating instructions

The first time you use the application it will be necessary to create a directory so that the sequence of events will be: Load part one by typing `RUN "PHONEHOME"` <**ENTER**>, press <**A**>. The program should then load part two of the application. Once loaded press <**C**>.

CPC 464 users: A very special word of warning: when using either programs or data stored on cassette, always ensure the tape is rewound to the position on the tape where the program or data starts, before attempting to load it into the CPC 464's memory.

Creating a directory

When entering names and numbers type the first name and press <**ENTER**>, type the surname and press <**ENTER**>, type the telephone number and press <**ENTER**>.

Once the user has created a directory, it will be possible to load part one and then proceed directly on to part three of the directory to use the facilities for looking at the various names and numbers.

Section B
Familiarity Breeds Confidence

Chapter Three

BASIC tools of the trade

As a result of working through Chapter 2, it is likely that you have now become familiar with a number of the words (i.e. commands and statements) that make up the language programmers use to give instructions to the CPC 664/464. This language is known as BASIC, or more precisely B.A.S.I.C., which is an abbreviation for Beginners All purpose Symbolic Instruction Code. As in all languages there is a very definite structure: each word not only fits neatly into a certain category, but also has a specific format, which is referred to as the syntax of the command.

By working our way through the three programs that make up the telephone directory application, it is now possible to consider in turn the consequence of each word, and its syntactic requirements.

Part one, the option menu, is useful now for several reasons. The first three lines all begin with the statement **REM** which is an abbreviation for the English word remark; it instructs the CPC 664/464 not to read the rest of this line, as it is merely a line of comments and notes for the programmer to understand various elements of the program, e.g. line 10 informs us of the purpose of the actual application, line 30 explains the purpose of the first program out of the suite of three.

```
10 REM Personalised Telephone Directory
30 REM Part One          Option Menu

40 CLS : PEN 3

50 FOR display=1 TO 22
60 PRINT TAB(display) "Telephone Directory"
70 NEXT
```

Line 40 illustrates the use of commands that have an immediate effect, commands in the true sense of the word, orders that are carried out immediately. These types of words are always very useful while working in direct command mode, as you will remember from Chapter 1.

The block of lines from 50 to 70 is very important, illustrating one of the most fundamental capabilities of any computer, i.e. being able to repeat a specific task a given number of times. The **FOR** statement sets up a variable (see Chapter 5 for details of variables) to which in this case the programmer has given the name 'display'. The contents of this variable initially is the number 1. Line 60 instructs the CPC 664/464 to print on the first row at the top of the screen, one column in from the left hand side of the screen, i.e. display=1. Line 70 instructs the CPC 664/464 to go back to the last line with a **FOR** statement in it, in this case line 50, and the contents of the variable 'display' is increased by one to the value of two. Line 60 now prints on the second row down as the cursor moves to a new line/row after each print command is executed. In addition, as the variable 'display' is now equal to two, the CPC 664/464's cursor begins printing two spaces in from the left hand side of the screen. The whole process is repeated until the contents of the variable 'display' is greater than the second number specified in the **FOR** statement, i.e. once display=22 this block of program is terminated, and the command on line 80 is executed.

There are two important ways to adjust such a block of program in order to count upwards in quantities greater than one. The following will cause an increase of five each time the block of program is repeated:

```
50 FOR display=1 TO 22 STEP 5
```

The following causes the **FOR—NEXT** block of program to count downwards in steps of one:

```
50 FOR display=21 TO 1 STEP -1
```

There is just one optional extra if as a programmer you find it easier to trace which **NEXT** statement ties in with which **FOR** statement. The **NEXT** statement can be written thus:

```
70 NEXT display
```

The block of program from line 80 to line 200 of the option menu creates two areas on the screen, changes the colour of the printing, and prints up essential information for the program user, suggesting that a certain response is required from him or her. The block of program from line 210 to 250 validates the response, ensuring the CPC 664/464 only responds to one of the two choices open to the program user.

The two lines 210 and 250, just like a **FOR—NEXT** statement, combine to ensure a repetition of a certain block of program. The line:

```
210 WHILE K$<>"A" AND K$<>"B"
```

reads 'While the variable named K-string does not have as its contents the capital letter A and the capital letter B, the program must execute the block of program up to the appropriate **WEND** statement, i.e. line 250.' The expression contained after the **WHILE** command on line 210 can of course vary according to the requirements of the program.

The command **INKEY$** on line 220 (read as in-key-string) allows the program user to make his or her choice of the options available by giving the variable **K$** some contents, a single character from the keyboard.

The function **UPPER$(K$)**, line 230, is a very special facility ensuring that it does not matter whether the program user has the CPC 664/464 in upper or lower case mode, as the contents of the variable **K$** will be automatically changed into upper case letters, i.e. capital letters.

Lines 260 and 270 of the option menu illustrate a fundamental capability of all computers, that of making a decision based on a specific condition, i.e. the contents of a given variable or variables.

```
260 IF K$="A" THEN RUN"Create"
270 IF K$="B" THEN RUN"Use"
```

Line 260 reads 'If the variable K-string has as its contents the letter A then and only then will the data recorder or disk system load and then execute the program which has been stored on cassette with the program filename of "**Create**".' Line 270 reads similarly, but with a different variable contents and a different program filename.

```
280 END
```

The last line of part one is merely a convenient way of terminating a program, in this case very much just for neatness of programming, particularly as lines 260 and 270 will ensure a direct continuation on to the execution of another program.

Let us now scan through the tools of BASIC that have been used in parts two and three of the telephone directory application to establish the various commands available to the program creator. All of these can be fitted into the structures shown by the use of the part One option menu program in the preceding pages of this chapter. These structures being the use of commands which take immediate effect, an appropriate classification term is *sequence type instructions*, as they are merely executed before the computer moves on to the next instruction/operation. Repeating a task several times has been shown by the use of **FOR—NEXT** and **WHILE—WEND** loops; the typical classification term for these is *repetition* or *iteration*. The third classification is the decision elements of creating a computer program, where the machine can compare any given condition to the state of a variable or variables, i.e. contents of a variable, and respond accordingly, illustrated by the **IF—THEN** commands. The classification term most commonly used is *selection*, the computer performing a specific operation as the result of a known condition being fulfilled.

GOSUB—RETURN is probably the most useful combination of command words available to the CPC 664/464 program creator who requires to develop professionally well structured programs. The command would read 'Go to the subroutine at line number'. The **REM** statement, which combines with the **GOSUB**, should explain the subroutine's objective. Lines 120 to 190 show the fundamental control unit of part two of the telephone directory application; thus the **GOSUB** command instructs the computer to divert the execution of instruction lines to the one specified, and to continue from that line number until the command **RETURN** is encountered.

```
120 GOSUB 210        :REM Option Menu
130 IF k$="A" THEN GOSUB 310
140                  :REM add to directory
150 IF k$="C" THEN GOSUB 620
160                  :REM create directory
170 GOSUB 960        :REM save directory
```

```
180 GOSUB 1190      :REM use or finish
190 END

210 REM Option menu on screen
220 LOCATE 1,2:PRINT"Choose the facility you require:"
230 LOCATE 5,4:PRINT"<C>reate a NEW directory"
240 LOCATE 5,6:PRINT"<A>dd to an OLD directory"
250 LOCATE 1,8:PRINT"Press either key <C> or <A>"
260 WHILE k$<>"A" AND k$<>"C"
270 k$=INKEY$
280 k$=UPPER$(k$)
290 WEND
300 RETURN
```

On the execution of a **RETURN** command, the computer diverts the execution of instruction lines to the line directly after the initial **GOSUB** command. In this way the section of program from line 120–190 can control which tasks the program user requires to be carried out by the appropriate selection of one of the program's subroutines. The second block of program lines, 210 to 300, illustrates a typical subroutine, beginning with a **REM** statement, defining the subroutine's objectives in order to make the reading of the program easier. The task is performed by the execution of the succeeding program lines. The subroutine is terminated by the **RETURN** command.

As in the instruction lines 130 and 150, both **GOSUB** and **RETURN** commands can be executed as a result of a known condition being fulfilled.

LOCATE and **PRINT** statements can be combined to produce some very interesting and professional screen displays.

A brief mention should be made here (for further details you should read Chapter 6) that the CPC 664/464 has a special feature whereby, using the syntax **PRINT** #[a number from 0 to 7], it is possible to position up to 8 text cursors at any one time, each cursor being known as an output stream. Initially they all reside at the topmost left of the screen. If the **#number** is omitted the default or number zero cursor is assumed.

The command **LOCATE** is followed by the optional stream **#number**; the syntax then consists of two numbers separated

by a comma. The first of these represents the number of columns from the left hand margin that the printing is to start; the second number represents the number of rows down. In the above example the stream has been omitted for clarity; it is therefore assumed to be zero. The maximum number of rows and columns is dependent on the screen **MODE** the computer is currently in, i.e. 0, 1 or 2.

By combining the **LOCATE** and the **PRINT** command, as in the example, reasonably professional screen displays can be obtained.

PRINT SPC(number) and **PRINT SPACE$**(number) are further means of arranging the screen display by giving a number of columns in from the left hand margin to print spaces. Essentially both do the same job, but as far as the computer is concerned the end result is attained from two different directions. The command **SPACE$** allows you to create a variable with the contents of a given number of blank spaces. For example, if **F$=SPACE$(5)** each time **F$** is printed by a **PRINT** statement it will produce five blank spaces. The **SPC** command acts more like a **TAB** command, giving a start position for the **PRINT** command.

The command **WINDOW** is a very specific instruction. By careful manipulation of the four numbers that follow the **WINDOW** instruction the programmer can determine the next area or section of the screen at which the succeeding print statements will be displayed. Again this command, just like the print statement, can benefit from the CPC 664/464's capability of having eight different cursors available for print detail on the screen. This in fact means you could conceivably create eight windows on the screen and use a different cursor to print information in each of them. Throughout the telephone directory suite of programs I kept, for the purpose of simplicity, to the default stream or cursor, and therefore omitted the #number. The complete syntax would be:

WINDOW#number,left side,right side,top edge,bottom edge

For the aspiring programmer, the use of ASCII codes will become more and more important. An ASCII code is a number which enables the computer to code the information coming in to it from the keyboard etc. The program lines in Figure 3.2 use

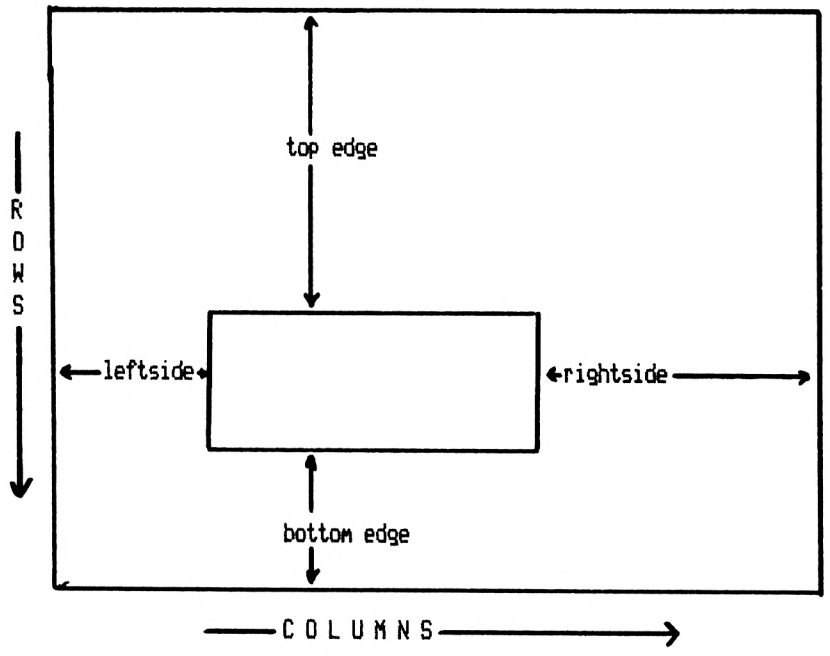


Figure 3.1 Illustration of the syntax of the **WINDOW** command

an ASCII code number to decide whether the <**SPACE BAR**> has been pressed. Here the instruction **CHR$(32)** relates to the code that the computer generates when the space bar has been pressed. Try this direct command: **PRINT CHR$(69)** and press <**ENTER**>. What happens? A letter **E** should

```
690 IF k$=CHR$(32) THEN PEN 2
700 IF k$<>CHR$(32) THEN PEN 5
```

Figure 3.2 The use of an ASCII code number within program lines

appear on the screen. In fact a different character is generated by placing in the brackets any number between 33 and 255 inclusive. The code **CHR$(32)** is in fact the computer's code for a blank space, which is so very important for separating commands, etc. Remember the computer treats a blank space just like any other character. Now experiment, starting with the following examples:

```
5 MODE 0
10 PEN 6
```

```
20 FOR c=1 TO 800
30 LOCATE 10,10
40 PRINT CHR$(225)
45 LOCATE 10,10
50 PRINT CHR$(224)
80 NEXT
90 LOCATE 1,1
100 FOR c=1 TO 100
110 PRINT CHR$(207)+CHR$(206)+CHR$(217)+CHR$(218);
120 NEXT
```

Try writing a short series of program lines, which will include a **FOR—NEXT** loop, so that you will be able to examine all the ASCII code numbers.

It is necessary to explain briefly the programmer's nightmare which occurs when the instruction **GOTO** 'a given line number' is used without very careful thought. It can tie the program up in knots so use it only after an **IF—THEN** statement, e.g.:

```
IF account<100 THEN GOTO 90
```

Other commands and instruction words will be dealt with in much greater depth in the appropriate section of this book. In short, the family of commands **INPUT** and **INPUT LINE** are concerned with enabling the program user to put information or detail into the computer's available memory. In the case of the telephone directory program these commands are used prior to the storing of the information on data cassette or disk, using the group of commands **OPENOUT**, **PRINT#9** and **CLOSEOUT**. The commands **OPENIN**, **CLOSEIN**, **INPUT#9** and **EOF** are concerned with retrieving data from a data cassette or disk which is an essential element of any useful application program.

Lines 1120 to 1200 demonstrate the format or syntax of using program lines to place or store data onto a cassette tape or floppy disk as storage of a data file:

```
1120 OPENOUT"data"
1130 c=0
1140 WHILE c<record
1150 PRINT#9,name$(c)
1160 PRINT#9,surname$(c)
1170 PRINT#9,phone$(c)
```

```
1180 c=c+1
1190 WEND
1200 CLOSEOUT
```

Lines 280 to 340 demonstrate the syntax of using program lines to retrieve data stored on tape or disk into the available memory space in the computer.

```
280 OPENIN"data"
290 WHILE EOF=0
300 INPUT#9,name$(counter),surname$(counter),phone$(counter)
310 PRINT counter,
320 counter=counter+1
330 WEND
340 CLOSEIN
```

Chapter Four

Debugging programs – editing facilities on the CPC664/464, and how to use them

Use the program loading facility `RUN"PHONEHOME"` to load and run the option menu program. Choose option A to load and run the program named 'Create'. Once loaded press the <**ESC**> key until the message `'Break in 270'` appears on the screen. Type `MODE 1: LIST` and press the <**ESC**> key. The program known as 'Create', i.e. part two of the telephone directory application, will scroll its way down the screen. Press the <**ESC**> key and the computer will stop scrolling; press any other key and it will restart scrolling the program listing. Press the <**ESC**> key twice in succession and the scrolling of the program listing will be terminated and the message `*BREAK*` will appear on the screen. The CPC 664/464 will be in direct command mode waiting for further instructions from the operator of the machine. Experiment with the syntax of the `LIST` command by using the following alternatives:

```
LIST 100-150     <ENTER>
LIST -100        <ENTER>
LIST 150-        <ENTER>
```

The number(s), in each expression relate to the line number of the program currently held in the computer's memory.

One of the most frustrating problems with beginning to program your own computer, is the constant problem of error messages appearing on the screen which prevents a successful conclusion being reached. Yet on the other hand, the solving of

program errors, or in technical jargon debugging a program, can be personally a very satisfying process and certainly a very useful way of developing a sound understanding of the underlying principles of programming a computer. The task is to be thought of as a puzzle for which a different kind of thinking is required to find the solution. It must be said though that it is at this stage most people give up the task. The only advice I can give is be patient, be prepared to experiment with various formats and most importantly experiment with your *ideas* of how the problem might work.

The logical order of operations when debugging a program currently held in the CPC 664/464's memory is:

1) `RUN` the program through to the point of the first error message, note the type and line number.
2) Type `MODE 1` <**ENTER**>.
3) Type `EDIT` (line number of line with error in it).
4) Check against program listing that there are no typing mistakes. If an error is found place cursor at the end of the offending error using the left/right arrow keys and press the <**DEL**> key the appropriate number of times. Now retype the word or phrase as it is meant to be.
5) Press the <**ENTER**> key.
6) Type `RUN`, press the <**ENTER**> key.

If this does not solve that particular error, you must now look closer at the type of error it is, for example, messages such as 'Unexpected `WEND`' suggest that a line is missing from the program that contains the matching `WHILE` command. A message '`NEXT` missing' suggests a missing line containing a `NEXT` command later in the program after a `FOR` statement. These types of errors will require the examination of a number of program lines on either side of the offending error line.

One of the most common errors is that of 'Syntax error', usually as a result of not leaving a blank space directly after a command word.

The program will not respond with an error message, but will not be doing what was planned if, for example, a variable given the name 'display' in a subsequent line is changed by a typing error to say 'dsly': a different variable has been created, which actually has zero contents, with the resulting effect on

the program. The term 'variable' will be explained in the next chapter.

To summarise, the usual errors, when copying into your computer from a printout of a program listing are as follows:

a) Typing differences within a specific line of program instructions.
b) Not ensuring that the appropriate spaces preceed each of the computer language's command words.
c) Missing out or not typing in a command word.
d) The omission of a complete line of program instructions.
e) Forgetting to press the enter key at the end of each line of instructions (a very easy error to make).
f) Not typing the line number at the beginning of a line of instructions, i.e. the line will have been entered in direct command mode and will have effectively been left out of the listing.

Looking to the future now, when you are creating your own programs the problems of debugging will have a new angle to them. In addition to syntactical or errors as result of clumsy fingers, there will be errors in the logic of the program's structure, and omissions or oversight as a result of being new to the game. I have always found the best way of getting around an error message is to examine the listing carefully, talking your way through the instruction lines, one at a time.

Methods of altering program lines after they have been entered in memory

Below are listed the simplest and most straightforward methods of altering lines of instructions held in the computer's memory. Make sure you are fully conversant with the `LIST` command and all its capabilities.

1) Retype the whole line of instructions again, including the line number.
2) Use the command `EDIT` 'line number', move the cursor to the offending error by using the left and right arrow keys, then use a combination of the <**CLR**> and <**DEL**> keys

to remove the error. Then type in the correct format. The computer is set into automatic insert mode so there is no need to create a space for your new command or additional instructions.

3) Hold down the <**SHIFT**> key at the same time as pressing the up arrow cursor key. The yellow square cursor will reveal a second cursor which, when you press the <**COPY**> key, will copy all the text that the second cursor is passing over onto the screen at the position of the first cursor. In this way it will be possible to copy those parts of the instruction line that were previously correct and add further instructions if need be. Once you are satisfied with the program line press the <**ENTER**> key and the cursors will be reunited together, one line below the last instruction line.

It will always be a good and fruitful exercise to type programs into your computer, as long as it is done with care and attention to what the program is doing and how it is doing it. Examine programs carefully and, by using the editing facilities, make changes and note the effect they have on the **RUN**ning of the program. Remember to make a copy of the program on cassette or disk before you make any (possibly disastrous) alterations. If you do happen to make changes that make the program irretrieveable, simply reload the original program from the copy you made initially.

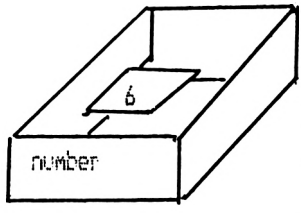
Section C
The Principles of BASIC

Chapter Five

Variable names and labels

During the process of looking at the direct command mode facility and the telephone directory as an application, it has been necessary to give passing reference to the concept of using variables to store information vital to the computer operator's purpose at that particular time.

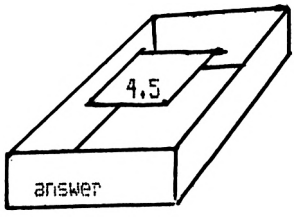
```
LET number=6
```



Contents is put into the the box, in this case the number six.

A name is given to the box or variable, so that it can recalled directly by its name.

```
LET answer=(12+6)/4
 ie answer=18/4
          =4.5
```



The contents is the result of a short calculation

The answer to the sum can now be recalled by asking the computer to print the variable known as 'answer'

Figure 5.1 The contents of a variable represented as the contents of a box

The most explicit way of viewing these stores of temporary information, known as variables, is by visualising a box within the computer's memory which has to be given a name/label so that its contents can be looked at, used or compared with the contents of other boxes, during the execution of, for example, program instructions by referring to each particular variable by its allotted name.

For a simple illustration of the use of such variable names, try the following direct command examples:

```
LET a=12:PRINT a <ENTER>
LET number=34:PRINT number <ENTER>
```

A variable name can consist of any number of characters, alphabetic or numeric, up to a maximum of forty, as long as it is a continuous string with no blank spaces. It must also begin with a letter rather than a number.

The **LET** statement is such a common occurrence as a command in the BASIC language that most computers do not require its use, i.e. it is an optional word while instructing the computer to create a box with some contents. For example:

```
number=56.67 <ENTER>
```

If you now instruct the CPC 664/464 to **PRINT** the contents of the box named 'number' it will respond by printing the number 56.67, on the screen.

Further examples of the use of variables in direct command mode:

```
wholenumber%=12.34:PRINT wholenumber%
<ENTER>
```

The CPC 664/464 will respond by printing the number 12 on the screen. The % sign signifies that the particular variable 'box' has a contents purely of whole or integer numbers.

```
wholenumber%=12.79:PRINT wholenumber%
<ENTER>
```

The CPC 664/464 will respond by printing the number 13 on the screen.

When using the % symbol as a suffix to a variable name, the CPC 664/464 will automatically round off the contents of the box with that name to the nearest whole number. This means

if the decimal places are less than 0.5 the number will be adjusted to the nearest whole number below. If the decimal places are above 0.5 then the variable contents will be adjusted to the nearest whole number above or greater than the real number.

```
aname$="Structured Programming":PRINT
aname$ <ENTER>
```

A box, labelled with a name suffixed by a **$** symbol, can have as its contents any string of characters that have been delimited by the use of quotation marks.

```
PRINT ANAME$ <ENTER>
```

You will notice that the same response as with the previous instruction will result. This is because the CPC 664/464 does not distinguish between upper and lower case variable names. This fact should be kept in mind for two reasons:

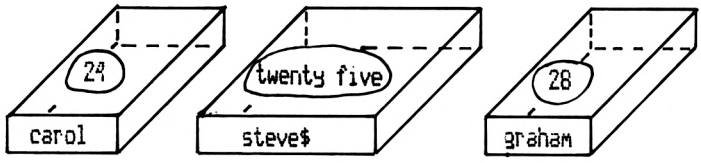
1) You cannot create two *different* variables with the same name but merely in a different character case.
2) You must always use the same upper or lower case for the variable throughout a program.

The arithmetical operation on lines 60 and 70 (Figure 5.2) requires some explanation of its function. The function **MOD** responds with the remainder of dividing the first number by the second, i.e. 24 divided by 3=8 and *no* remainder. The function \ responds with the integer answer, having divided only by the whole number, leaving off any decimal points, i.e. 28/3reads 28 divided by 3 is 9 and 1 remainder, the integer division answer is 9.

The series of characters represented by **steve$** remains constant and merely prints the string of characters 'twenty-five'. Try the following command:

```
PRINT steve$+steve$ <ENTER>
```

The result should be 'twenty-fivetwenty-five'. Note that there is no space between the two strings of characters because the instruction is literally to lump these two string variables together as one. The addition sign when used in conjunction with strings of characters has a different function to that when used with numerical expressions.



The instructions to the computer would be read as follows:
LET the box, named carol, have as its contents the number 24
LET the box, named steve$, have as its contents the string of characters 'twenty five'
LET the box, named graham, have in it the number 28

```
The aritmetic:                  The contents of:
24+28=52                        carol+graham=52
24 28=672                       carol*graham=672
28 24=1.16666667                graham/carol=1.6666667

24 MOD 3=0                      carol MOD 3=0
28 \ 3=9                        graham \ 3=9

                                steve$=twenty five
```

The CPC464\664 has to be instructed in a very precise manner, it can then perform the same arithmetical and manipulative operations as above.

```
The Program                             The Screen Display on execution
                                        of the program
10 carol=24:steve$="twenty five"
20 graham=28
30 PRINT carol+graham                   52
40 PRINT carol*graham                   672
50 PRINT graham/carol                   1.16666667
60 PRINT carol MOD 3                    0
70 PRINT graham \ 3=9                   9
80 PRINT steve$                         twenty five
RUN
```

Figure 5.2 Demonstration of the use of variables as stores of information

You will appreciate as you become a proficient programmer, that variables are the mainstay of all programs. The use of variables is very widespread, because they can do extremely neatly a number of valuable tasks. Let us now survey the telephone directory suite of programs to examine the use of variables *in situ*, as it were.

Part one: The option menu

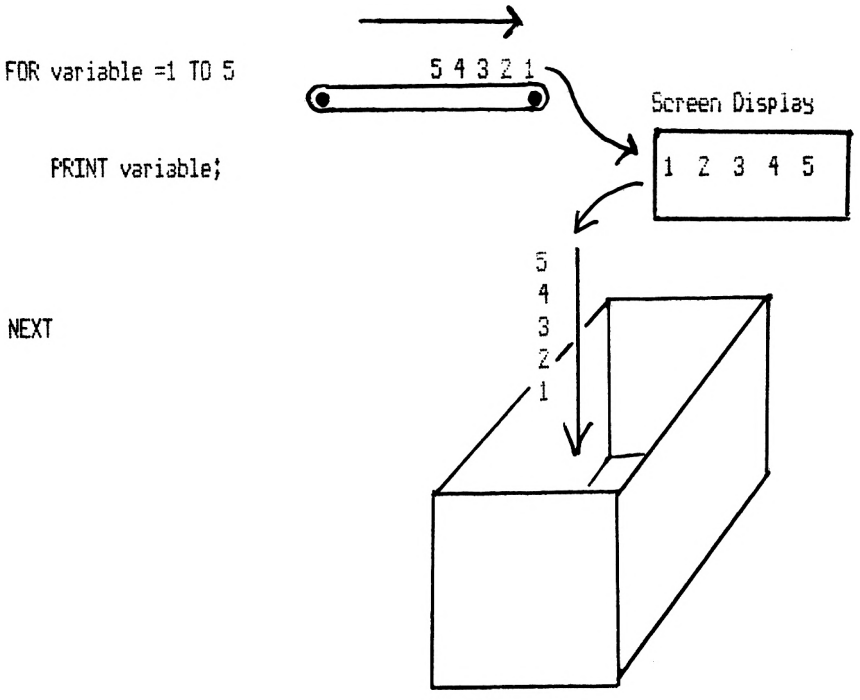


Figure 5.3 The use of a counting variable—the **FOR—NEXT** loop

The first variable we encounter is a very special one, being an integral part of a **FOR—NEXT** loop. The contents of the variable named 'display' is repeated, increasing by one whole number until 'display' has a contents of twenty-two.

The option menu program lines 110 to 120 and 210 to 270 illustrate one use of a variable.

```
100 PRINT"Do you require to:"
110 PRINT:PRINT"<A> Create or add to your directory "
120 PRINT:PRINT"<B> Use your directory"

210 WHILE K$<>"A" AND K$<>"B"
220 K$=INKEY$
230 K$=UPPER$(K$)
240 PEN 1
250 WEND
260 IF K$="A" THEN RUN"Create"
270 IF K$="B" THEN RUN"Use"
```

The variable illustrated, **K$**, neatly links the action asked for and displayed on the screen for the program user to the decision processes that are carried out within the computer's memory. By the use of the command word **INKEY$**, the contents of the box named **K$** is determined by the program user. If the user presses the <**A**> key the program called '**Create**' will be loaded and executed from the tape or disk, as appropriate.

Part two: Creating a telephone directory

The following three program lines demonstrate a typical starting point for a majority of programs: the setting up of variables and the placing of contents within them.

```
70 t=100:counter=0:record=0
80 DIM name$(t),surname$(t),phone$(t)
90 f$="Firstname":s$="Surname":ph$="Phone No."
650 PRINT f$,s$,ph$
```

Line 70 sets up the numerical variables, **t** being the maximum number of telephone numbers/names to be stored in the program. The variable **counter** will count the number of records, and various other counting tasks throughout the program that will begin at a value of zero. The variable **record** will print on the screen the current record being entered by the program user.

Line 80 illustrates a special form of variable known as a *dimensional array*, which will be discussed in detail in Chapter 14. Simply, dimensional arrays are a range of variables that are all linked in some way, yet they all require a unique way of being identified. Therefore the variable name/label will be constant but the number inside the brackets will be unique. In this case 101 different boxes have been created by the use of dimensionally assigning names(**t**), with **t=100**. Thus: **name$(0),name$(1),name$(2)**... and so on to ...**name$(100)**. The variables could, if desired, be two-dimensionally arrayed thus: **name$(100,100)**, creating a lattice work of coordinates, i.e. **name$(0,1), name$(0,3),name$(2,5)**. There are 10 000 variables

which could have been created if this method of allocation had been used.

The program line 90 presents a useful technique of creating a variable consisting of a word that will be used and repeated throughout the program. Whenever the word is required time and effort is saved by merely instructing the computer to print the contents of the appropriate variable, as illustrated in program line 650.

Another method of counting variables, is by the use of a WHILE—WEND loop. This form of loop requires a physical manipulation to make it count up or downwards, e.g. line 1150 in this example, until the condition in line 1110 is satisfied.

```
1100 counter=0
1110 WHILE counter<record
1120 PRINT#9,name$(counter)
1130 PRINT#9,surname$(counter)
1140 PRINT#9,phone$(counter)
1150 counter=counter+1
1160 WEND
```

Part three: Using the telephone directory

The final method of placing contents into a variable is by means of the INPUT command:

```
1030 PRINT n;name$(n),surname$(n),phone$(n)
1040 LOCATE 2,4: INPUT;name$(n)
1050 LOCATE 12,4: INPUT;surname$(n)
1060 LOCATE 25,4: INPUT;phone$(n)
```

Reacting to your CPC 664/464 – INPUT statements

So far, we have explored the capability of the computer to store information in the form of variables created within the program itself, hinting only at the capability of the operator/program user to place contents within a variable created from within the program. A computer can only be truly useful when it can take information from the operator/program user and process it, and provide further information as a result. The simplest way of doing this is for the computer to set up a variable box, where the contents is determined by the program user who responds to a question or series of options displayed on the screen. The response therefore becomes the contents of this particular variable.

A simple example might be:

```
10 INPUT "What is your first name";name$
20 INPUT "What is your favourite number";number
30 FOR display=0 TO number
40 PRINT name$,
50 color=display MOD 9
60 PEN color : PAPER color+1
70 NEXT display
80 PRINT " Have a nice day! "
```

This is a rather meaningless example, but it illustrates the important facts concerning the preliminary use of the **INPUT** statement. The rules for variable types and the suffix used are the same as if the information was being stored via a **LET** statement.

The **INPUT** statement is very much like a **PRINT** state-

ment, as far as displaying on the screen, at the position of the text cursor, anything the program creator places after the INPUT command and inside quotation marks, as in program lines 10 and 20 above. The one additional advantage is that it will then wait for the listed variable to be given some contents by the program user.

The CPC 664/464, have prompted the program user for some information, sets about processing it according to the programmed instructions, in this case a FOR—NEXT loop, the number of repeats of the printing and change of colour routine being determined by the favourite number of the program user. This type of use of input statements, setting up the direction and format of the program to follow, is used extensively and becomes essential if programs are to be used for many different user requirements. For example, a program may be designed to take telephone numbers, surnames and first names, but another user may require surnames, addresses, and telephone numbers. Both are essentially the same type of program, both have three fields of information, but the description of the fields varies. It is in this situation that a series of INPUT statements becomes very useful, to design the format and layout of just such a program application. The information for the format of the file and its layout is then stored on cassette data file, along with the personal details, ready to be used the next time the program application is required. This type of programming can be seen as creating the instructions at a level known as first principles, allowing the program user to personalise the programmed application for his or her own requirements.

A skeleton program for such a purpose could be as follows:

```
10 REM A information design application
110 counter=1
120 CLS:LOCATE 1,1
130 GOSUB 170
140 GOSUB 470
150 GOSUB 360
160 END
170 REM Create information
180 INPUT "A title for this screen of information  (max 30 characters)";title$
190 IF LEN (title$)>30 GOTO 180
200 INPUT "How many fields of information";field%
```

```
210 DIM detail$(field%)
220 PRINT
230 PRINT "When entering information, DO NOT use"; PRINT "commas."
240 PRINT
250 PRINT "When entering information, make sure words do not straggle the
end of one line and the beginning of another."
260 PRINT
270 PRINT "Press the <ENTER> key when you have"; PRINT "completed each entry."
280 PRINT
290 WHILE counter<=field%
300 PRINT "Field number";counter
310 INPUT "Enter details (max. of 255 characters)";detail$(counter)
320 PRINT
330 counter=counter+1
340 WEND
345 CLS
350 RETURN
360 REM information display on screen
380 PRINT TAB(5) title$
390 PRINT
400 counter=1
410 WHILE counter<=field%
420 PRINT detail$(counter)
430 PRINT
435 IF VPOS(#0)>=17 THEN GOSUB 470
440 counter=counter+1
450 WEND
460 RETURN
470 REM Press any key routine
480 LOCATE 1,24
490 PRINT "Press Any Key"
500 a$=INKEY$
510 IF a$="" THEN 500
515 CLS
520 RETURN
```

As it stands at the moment the program enables the user to design a series of sentences, words, etc. to be later displayed on the screen in standard layout. Add to this program the capability to save the parameters of the sentences or words onto a data file stored on cassette or disk. The application could be used to keep in an organised fashion information such as cookery recipes, step-by-step guides to doing various mechanical/maintenance tasks, all of which could then be

created and saved onto cassette/disk, to be recalled whenever the information is required.

We now come to the central feature of all good programs, that is, the ability of the program to interact with the program user. By identifying three stages to any program that is interactive with its user, the process of planning and creating a program will be simplified, to the extent of being merely a solving of problems in a logical order. The three stages that all programs will involve are:

1) The program user reacting or responding to information, usually displayed on the monitor's screen.
2) The program displaying information on the screen to enable the user to respond, usually by hitting a key on the keyboard.
3) The program containing within it lines of instructions which carry out the processes that the screen display instructs the program user can be carried out.

Consider these three points very carefully.

Points 2 and 3 are clearly the constituents of the program itself, combining so that the stated aims of the application can be achieved for the particular user of the program. By deliberately separating elements of the program it becomes apparent how it is possible to write programs that are clearly out of tune. The details displayed on the screen bear no relation to what is happening within the computer's memory. For example, the screen could say 'hit the <**SPACE BAR**>' and nothing happens. This would occur if the program only had output i.e. print statements, and did not contain the program lines to respond. A simple three line instruction will clear the screen, locate the **INPUT** statement, and print the prompt on the screen:

```
10 CLS
20 LOCATE 10,10
30 PRINT "Hit the <SPACE BAR>"
```

Unfortunately these instructions will not process any reaction from the program user as the program lines simply do not contain the capability to do so.

In order to demonstrate further such an approach let us now consider two examples from the telephone directory applica-

tion. Lines 110 to 540 below illustrate the relevant program lines. Both examples follow the same guidelines, the displaying of instructions for the program user to read, with the suggestion that he or she is then to respond accordingly. This is effectively achieved by the use of the **PRINT** statement.

```
110 WHILE k<>4
120 GOSUB 380    :REM select facility
130 ON k GOSUB 550,710,950,1230
140 WEND

380 REM select facilties
390 WINDOW 1,40,1,25: CLS
400 PRINT TAB(8)"Facilities Available"
410 LOCATE 5,3
420 PRINT"Choose each option by pressing the"
430 PRINT"appropriate number and hit the <ENTER>  key"
440 LOCATE 8,7
450 PRINT"<1> Browse through directory"
460 LOCATE 8,9
470 PRINT"<2> Select and search"
480 LOCATE 8,11
490 PRINT"<3> Amend a record
500 LOCATE 8,13
510 PRINT"<4> Quit
520 LOCATE 8,15
530 INPUT k
540 RETURN
```

This example then uses an **INPUT** statement to give the variable k contents of a number between 1 and 4 inclusive. In this case the subroutine is then completed, and the program control returns to line 130 for the response to be processed. This line is a special line of instructions that works very much like an **IF—THEN** statement. If the contents of k is 1 then the program control diverts to the subroutine beginning at line 550. If the contents of k is 2 then the program control diverts to the subroutine beginning at line 710 and so on. Thus immediately the three fundamental elements of a program have been fulfilled.

1) The program user has had to interact with the program.
2) The program has displayed information on the screen so that the user can make a response.

3) The program has been able to process the response and the required action has been fulfilled subsequently in the direction the program control has been diverted to.

It is now pertinent to throw in an additional point for consideration. What would happen if the program user were to respond with either a letter instead of a number or a number less than one or greater than four? Try it and see what happens! What should happen is that if you enter a letter and hit <**ENTER**>, the computer's response will be:

```
?Redo from start
```

This is an inbuilt facility of the CPC 664/464 to repeat an **INPUT** command if the program specifies numeric variable but strings of characters are entered instead. If a number less than one or greater than four is entered the screen will clear and the whole option menu will be reprinted on the screen. The reason is that lines 120 and 130 are positioned inside of the **WHILE—WEND** loop, lines 110 and 140. Line 130, the **ON—GOSUB** statement, will only divert the program control for the variable contents relating to the number of subroutines listed after the **GOSUB** command. So after an 'incorrect' user response the program merely passes to line 140 and therefore the repeat loop is executed once more, i.e. the option menu is displayed, awaiting a program user response.

What happens if a very large number is entered, say several hundreds of thousands? Unfortunately the computer's response is an error message that interrupts the program and returns the control to the user by showing the **READY** sign. This is a very frustrating problem and the only remedy as the program stands presently is to re**RUN** the program and load the **DATA** tape again. There is a remedy in adding this program line to your part three program:

```
35 ON ERROR GOTO 120
```

Remember that you will have to save the amended program onto tape or disk as appropriate, otherwise the advantage of this additional line will be lost. This program line enables the program to override the effect of the computer encountering an error, and instead of displaying a message in this case the part three option menu is displayed, waiting for a user response.

This avoids the need to reload the data from the **DATA** file tape or disk.

The procedure of making your programs user foolproof is often referred to as validating the program for the user. It is a fundamental program skill and worth considering in depth.

The second example of screen display and user response with program processing ability follows a similar format to that in lines 110 to 540 with an important difference in the way the program waits for the user to respond. Instead of using an **INPUT** statement linked to an **ON—GOSUB** statement it uses a combination of a **WHILE—WEND** loop with an **INKEY$** statement which, instead of actually waiting for contents to be placed in the variable thus named, merely places the last key pressed as the contents of the variable.

```
80 WINDOW 2,40,2,11
90 CLS:PRINT:PEN 2
100 PRINT"Do you require to:"
110 PRINT:PRINT"<A> Create or add to your directory "
120 PRINT:PRINT"<B> Use your directory"
130 PRINT
140 PRINT "Press the appropriate <key>"
150 WINDOW 2,40,13,21
160 CLS:PRINT:PRINT
170 PRINT "Remember you must have created a "
180 PRINT
190 PRINT "directory before you can use it."
200 PRINT

210 WHILE K$<>"A" AND K$<>"B"
220 K$=INKEY$
230 K$=UPPER$(K$)
240 PEN 1
250 WEND
260 IF K$="A" THEN RUN"Create"
270 IF K$="B" THEN RUN"Use"
```

By stating in line 210 that the loop is to be repeated until the contents of the variable **K$** is either 'A' or 'B' the program reads 'While k-string doesn't equal "A" or "B" repeat this section of program instructions'. The problem of validation has been solved merely by ensuring that the loop will be repeated until one or other of the two required responses is

made by the program user. In more complicated response sections line 210 could be extended to encompass more than just two options. Once one of the 'correct' responses has been made the program proceeds to carry out the loading of the next program from cassette or disk, the name of the program having been determined by the program user.

To complete this way of looking at application program writing consider the program lines 730 to 940 and decide which groups of lines would relate to

1) Screen display instructions.
2) Program waiting for the user's response.
3) Program processing instructions as a result of the user's response.

```
730 CLS
740 PRINT"Do you want to select by:"
750 PRINT:PRINT"<A> First name
760 PRINT:PRINT"<B> Surname"
770 WHILE k$<>"A" AND k$<>"B"
780 k$=INKEY$
790 k$=UPPER$(k$)
800 WEND
810 PRINT
820 IF k$="A" THEN INPUT"Which first name";name$
830 IF k$="B" THEN INPUT"Which surname";surname$
840 PRINT:PRINT "   ";f$,s$,ph$
850 WINDOW 1,40,10,21
860 c=0
870 WHILE c<record
880 IF k$="B" THEN 900
890 IF name$=name$(c) THEN f=1: PRINT c;name$,surname$(c),phone$(c)
900 IF surname$=surname$(c) THEN f=1: PRINT c;name$(c),surname$,phone$(c)
910 c=c+1
920 WEND
925 IF f<>1 THEN LOCATE 10,8:PRINT   No Record Found "
930 GOSUB 1280
940 RETURN
```

Now compare your answer to that given in Figure 6.1. Hopefully they will be the same or similar.

```
                                                    Screen displaying instructions
730 CLS
740 PRINT"Do you want to select by: "
750 PRINT:PRINT"<A> First name
760 PRINT:PRINT"<B> Surname"
                                                    Computer waits for valid response
770 WHILE k$<>"A" AND k$<>"B"
780 k$=INKEY$
790 k$=UPPER$(k$)
800 WEND
                                                    Computer processes response
810 PRINT
820 IF k$="A" THEN INPUT"Which first name";name$
830 IF k$="B" THEN INPUT"Which surname";surname$
                                                    Screen display for processed response
840 PRINT:PRINT "   ";f$,s$,ph$
850 WINDOW 1,40,10,21
                                                    Processing and display result of processing
860 c=0
870 WHILE c<record
880 IF k$="B" THEN 900
890 IF name$=name$(c) THEN f=1: PRINT c;name$,surname$(c),phone$(c)
900 IF surname$=surname$(c) THEN f=1: PRINT c;name$(c),surname$,phone$(c)
910 c=c+1
920 WEND
                                                    Press Any Key routine
930 GOSUB 1280
```

Figure 6.1 The subroutine divided into specific task blocks: Part three—search and select subroutine

While we are looking at this particular subroutine of part three of the telephone directory, refer back to the original listing and load the program into your computer. Use the <**ESC**> key to break into the program and `DELETE 720` <**ENTER**>. Now use the program several times, repeatedly responding to the menu for the search and select option. What happens once line 720 is removed? It is an important processing point: once the option has been used previously in that particular session, the program will work, but will not be doing quite what is expected!

Section D
From Little Blocks to Structured Programs

Chapter Seven

Structured planning on CPC 664/464 applications

Once again let us see the personalised telephone directory as a model, to guide us through the logical process of planning just such an application. First, remember that when creating and developing your own applications, the pen and paper work of planning the application is the first stage. This book has explained the process of creating a program from the situation of the whole first, so that is possible for you to examine the creation process from the creator's point of view. In other words, when you have your own CPC 664/464 application ideas you will visualise first exactly what you require, and then create it. The scene has now been set, the whole application has been examined and used, the parts must now be singled out for close up analysis.

The idea must come first and this is a thought process. In the case we are looking at the idea arose out of the need to hold a hundred phone numbers with the ability to browse through or select by surname or first name, and to remove that unsightly mess of scraps of paper around the telephone.

Once the idea is formulated, what will be the major tasks of the operator when using the application?

1) The creation of the initial directory.
2) The adding of new names and telephone numbers to a directory previously created.
3) The use of the directory in terms of browsing, selecting, and viewing the names and telephone numbers.
4) The editing of the names and numbers previously stored in the directory.

Among the tasks identified, 1 and 2 are clearly related and therefore could be written into a program together, as they will mutually use the same directory creation routines. The tasks numbered 3 and 4 are related, in as far as to be able to edit a record it is first necessary to select and view the record. It would therefore appear to be most appropriate to divide the application into two separate programs, one dealing with the creation of the directory and the other dealing with the use of the directory on a daily basis. The advantages are numerous; the main ones being the simplicity involved. By keeping the various tasks of the operator separate, the program creation is easier and simpler to handle, and while using the directory the reduced amount of program to be held in memory at any one time means the loading, i.e. start up time, will be greatly reduced. One of the frustrating elements of cassette storage is the time of program loading, even with the CPC 664/464's 'speed write' facility. Therefore any programming aid to shorten program loading time must be an advantage. This is obviously not a consideration if a disk system is being used. Once the decision has been made to develop a suite of programs they must be preceded with a short routine to act as a main menu for the operator to choose which program is required. The same routine can also be used as a title page for displaying the application's function and any essential information to ensure incorrect options are not chosen.

The programs that will be involved in the telephone application will be/are:

1) Option menu – Screen display options to load the appropriate program.
2) Create – Program to create and store the telephone directory.
3) Use – Program to use the directory on a daily basis.

Planning the screen displays

The option menu will be considered first. The usual plan is to produce a small graphic/illustrative title page followed by the essential instructions to start up the program facilities.

Screen displays for telephone directory application

1) Title page for option menu:

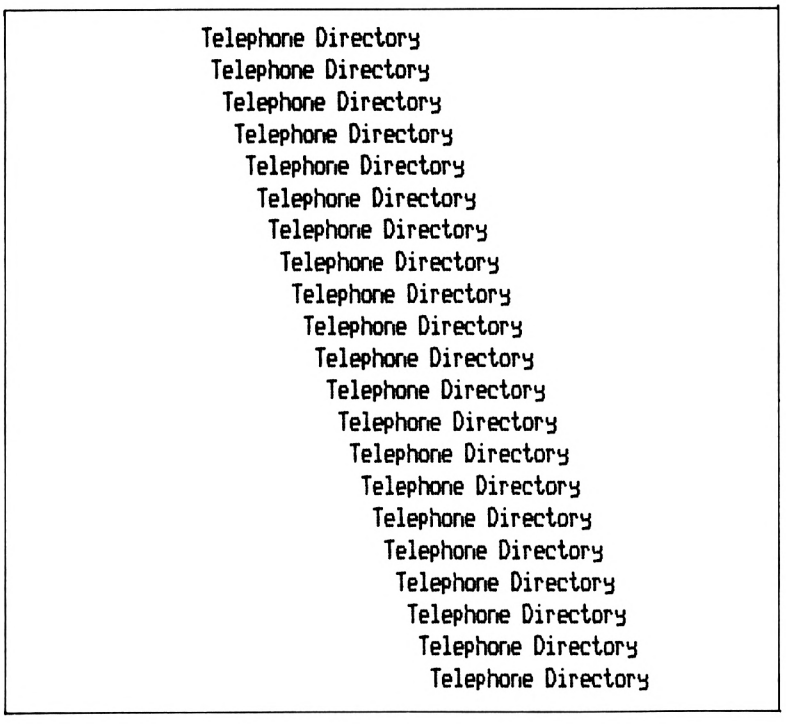


Figure 7.1 Initial screen display of part one: option menu

2) Essential information to start up the program facilities:

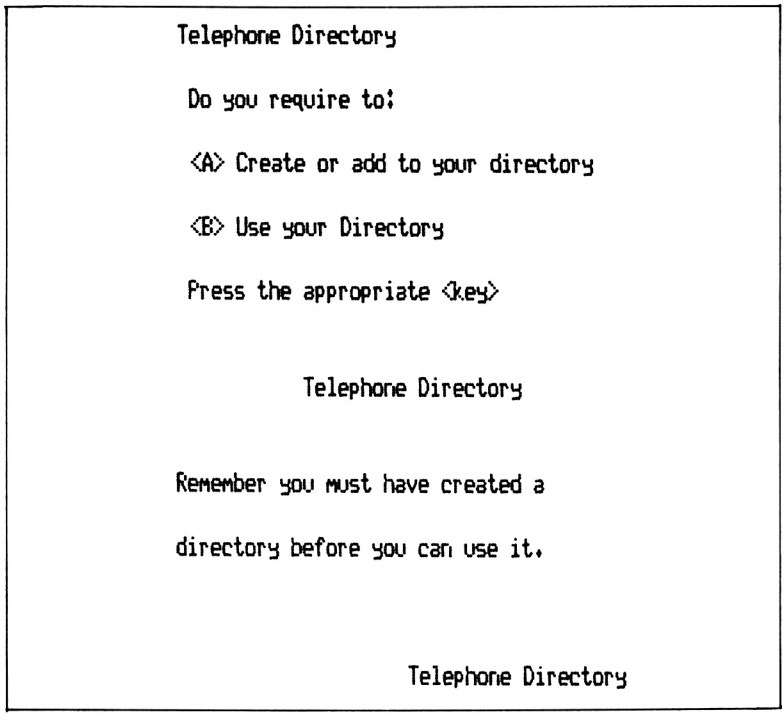


Figure 7.2 The options and initialisation instructions

3) Create or add to a telephone directory, a sub-menu for initialisation of part two of the suite of programs:

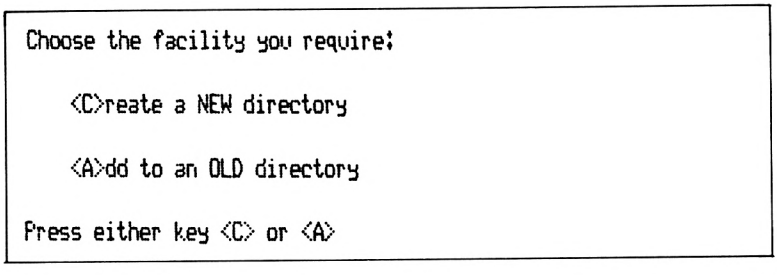


Figure 7.3 Part two offers two facilities—this screen display provides the details

4) Date entry screen for creating or adding to the telephone directory:

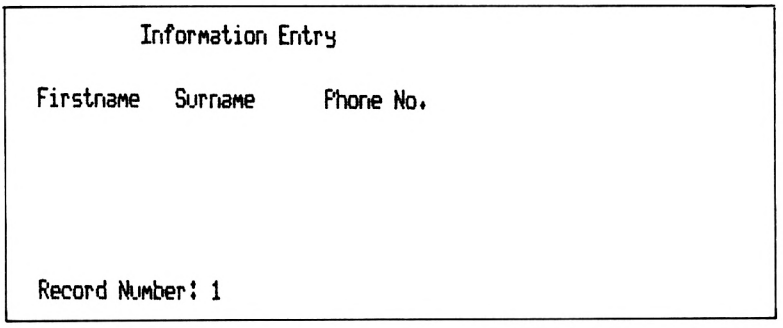


Figure 7.4 Part two enables the user to enter name/phone number information

5) The sub-menu to initialise the facilities available in part three of the suite of programs:

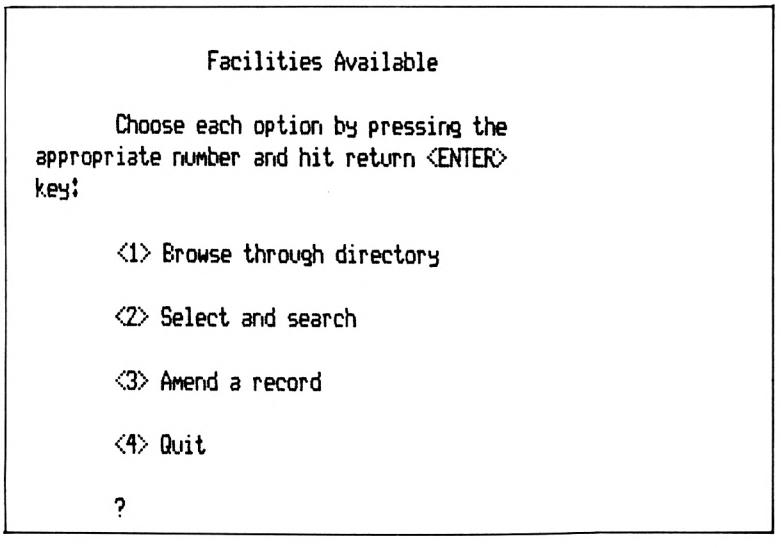


Figure 7.5 Part three provides four facilities—this screen displays these options

Sequence, selection and repetition

For the time being, let us leave the application we are considering and look at some essential theory that is required. It is now prudent to look at three terms that can be applied to virtually every action of any machine or living animal which performs tasks in order to attain a specific goal.

First we shall describe them, and then explain the ways in which we can apply these terms not only at the level of programming the CPC 664/464, but in addition apply them to tasks performed in our everyday life.

Sequence—A specific task/job is merely executed and the performer of the task then moves on to the next task.

Selection—A task/job has to be executed, but only if a given condition is fulfilled before the task can be initiated. For example, a question requiring a yes/no answer is posed; a yes response will result in one task being executed, a no response will result in another task being executed.

Repetition—A single task or a number of tasks, which can be a series of sequence or selection type tasks, are to be repeated until a specified condition is fulfilled. For example, a wheelbarrow is to be filled with earth using a small bucket and shovel; the bucket will be repeatedly filled with soil and emptied into the wheelbarrow until the wheelbarrow is full to a predetermined level.

We will now examine how these theoretical categories help in the task of creating a program.

Sequences of tasks as applied to everyday living and program creation

The concept of placing every activity in sequence is fundamental to the way in which most people organise their daily activities. In essence, if a series of tasks are placed in a sequence, a logical order is imposed upon their execution. For example you cannot make your bed before you get up out of it. You cannot watch TV until you have turned it on. Similarly, in terms of using a computer application program such as a telephone directory, you cannot use the directory to look up someone's phone number until you have actually gone through the process of creating the directory first. In terms of the program creator relating to the program user, the screen display providing the instructions must be displayed before the a response can be made. Therefore, referring back to the discussion in Chapter 6, the sequence of screen instructions/computer waiting for user response/computer processing and

acting on response is a logical sequence of events that would not make sense if executed in any other order.

Selection of tasks as applied to everyday living

Sequences become unrealistic in that they assume that there is no interaction with the present situation, merely repeating the same commands/instructions every time that particular sequence of events is called up to be executed.

Maintaining the same basic structure, but including a question where the subsequent events are determined by the answer to the question, enables a series of choices to be invoked and therefore a number of possible avenues to be explored. Selection is achieved by the process of deciding if a given or known condition is fulfilled. Then and only then will a particular sequence of events be performed. As an alternative to the a=b conditional decision, the decision can be based on several other conditional frameworks:

1) If A is less than or greater than B then perform this sequence of events/operations.
2) If A is NOT equal to B then perform this task, sequence of events or operations.
3) If a combination of conditions are required, e.g. equal to, not equal to, less or greater than, they can be considered by the use of the link words AND or OR.
4) AND means each and every condition must be fulfilled in order to perform the task or sequence of events.
5) OR means that only any one condition is to be fulfilled before the task or sequence of events is performed.

Repetition of tasks as applied to everyday living

In our everyday life we clearly repeat similar tasks hourly, daily, weekly, monthly, and quite possibly yearly. Therefore to talk in isolation of sequences of events, even if they do include conditional decisions, without the concept of ever repeating any task or sequence of operations, is not realistic. Essentially tasks have to be repeated if a given or known condition is not fulfilled, for example, most people go to school every weekday (excepting holidays) between the ages of 5 and 16. This forms a

sequence of operations that is repeatedly carried out until a condition is fulfilled. The opposite form of repetition is also valid and must be considered, namely, repeating a particular task as long as a condition *is* fulfilled. An example would be the instruction that while it is raining and you are out of doors wear a waterproof coat. An example at programming level would be while a `FOR—NEXT` loop is counting, the screen display will continue changing colour.

Repetition or, if you prefer, at programming level, loops of program instructions can be unconditional or more usually conditional on a state of affairs that will never be fulfilled. Clearly these loops will repeat the sequence of tasks 'for ever' or until the plug is pulled out. In most cases this situation is undesirable in computer program terms and should be considered carefully before creating a continuous loop within a program.

Imposing a structure on the nature of computer programs

By now you should be considering each and every program in terms of small blocks doing a specific task, for example:

1) Screen information
2) Waiting for a 'user' response
3) Processing/acting on the user response

Each block itself is constructed from a sequence of operations with a logical order imposed on it. To apply the block to the current situation it is implemented with regard to the fulfilment of several conditions which will determine the actual avenue the sequence of operations will take and at what points they will be repeated and hence terminated.

Each block will also be part of a much larger structure that can be given a similar structure of sequences of blocks that, as a result of conditions imposed, will follow different avenues according to the user's responses, as well as repeating various combinations of program blocks, to fulfil the stated aims of the program itself.

Chapter Eight

Diagrams make the mind clearer

When we think of computer experts using diagrams to illustrate their programs the term flow chart springs to mind and we immediately imagine great unwieldy squares, rectangles, and diamond shapes, with words written inside them, all linked by a spaghetti of lines directing the user up, down, across, and around the diagram. There is a definite concern that it is not really very logical in its final form.

Here I am going to suggest an alternative approach to drawing a diagram that can be followed from the top of the page across and down, just as if you are reading a page of text. A top-down approach to programming is currently a popular technique and this form of diagram fits the requirements for just such an approach.

Examine Figure 8.1 and try to identify *a*) sequence type tasks, *b*) selection type tasks, and *c*) repetition type tasks. Start reading at the start oval, travel down the vertical line until a horizontal line is met, follow the instructions given, moving horizontally. On completion of the horizontal line, travel back to the last vertical line, and continue travelling downwards to the next horizontal line of instructions. Follow them across and down until all the instructions have been executed, dependent on the meeting of given conditions and responses to the decisions that have to be made.

Figure 8.1 illustrates the potential of top-down flow charts. Note that it is possible to use the approach for everyday events as well as at programming level.

Let us now break down the various diagrammatical elements by examining the following four Figures 8.2, 8.3, 8.4 and 8.5.

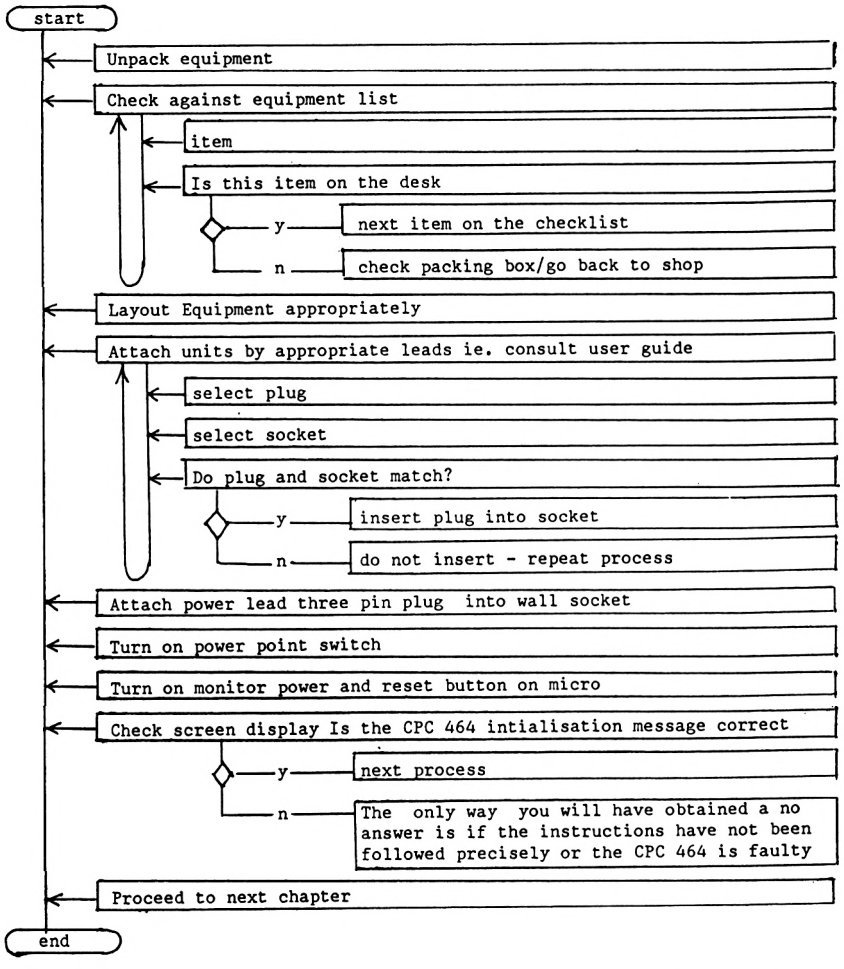


Figure 8.1 Using a structured diagram to set up your CPC 664/464

Sequences of operations

The form for illustrating a sequence of operations is very similar to the way they would be read. Do this operation, then this operation and so on until the task is completed. There is no consideration given to variation in conditions at any particular time. It is assumed that these tasks are completed in the same manner each and every time the sequence is implemented. These diagrams are essentially summaries of events. Not until we begin to use program language terms will

we see the relevance of purely sequence type operations, for example, direct command type instructions in Figure 8.6.

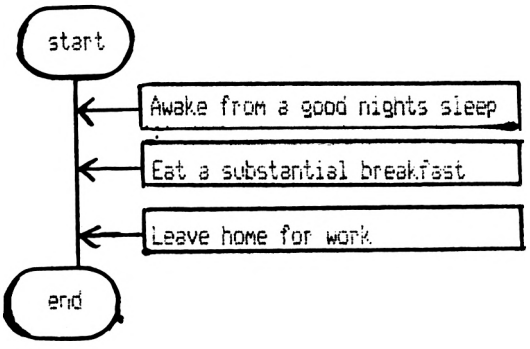


Figure 8.2 Illustration of a sequence of operations, imposing a logical order onto them

Selection of operations

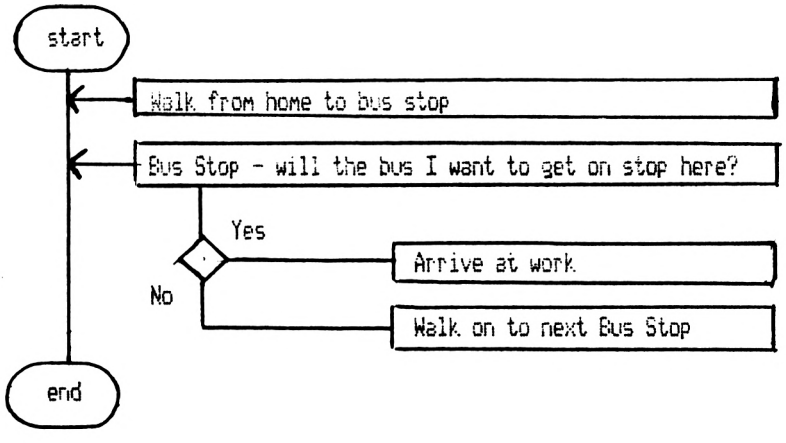


Figure 8.3 An explanation of a sequence of operations becomes more realistic if built into it is the capacity for choice or conditional decisions. Selection of one or more avenues a task may take can be easily represented by this type of diagram

To provide a choice means that the user has to take one or other of two routes on completion of whichever option is chosen. In this example, Figure 8, the task is completed by returning to the vertical line, the next statement being the end command. If one now considers this proposition carefully it is

not realistic: what would happen if the next bus stop was not the one where a bus will stop to take the person to work? It is painfully obvious that the choice segment of the task will have to be capable of being repeated until a suitable bus is boarded by the person travelling to work. The dropped U-shape in Figure 8.4 demonstrates this inclusion which provides a touch of further realism.

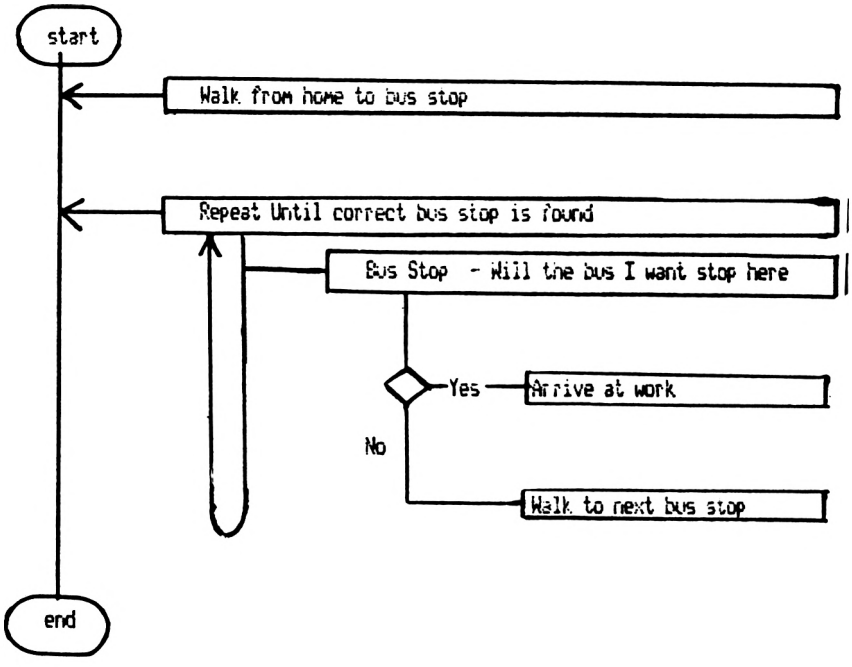


Figure 8.4 Repetition of events, including a decision-making process

Developing the concept of repetition

If a certain outcome is needed before progressing to the next set of operations, a method of repeating the task is required. The first set of operations in Figure 8.5 demonstrates this point.

This example illustrates the concept of multiple choice with options being repeated throughout one's life at various times, one or the other being followed as a set of operations. It is at this point that the use of a series of symbols can permit the development of further detail in subroutines. The symbol tells

the user that part of the operation will be documented elsewhere, labelled with the appropriate number.

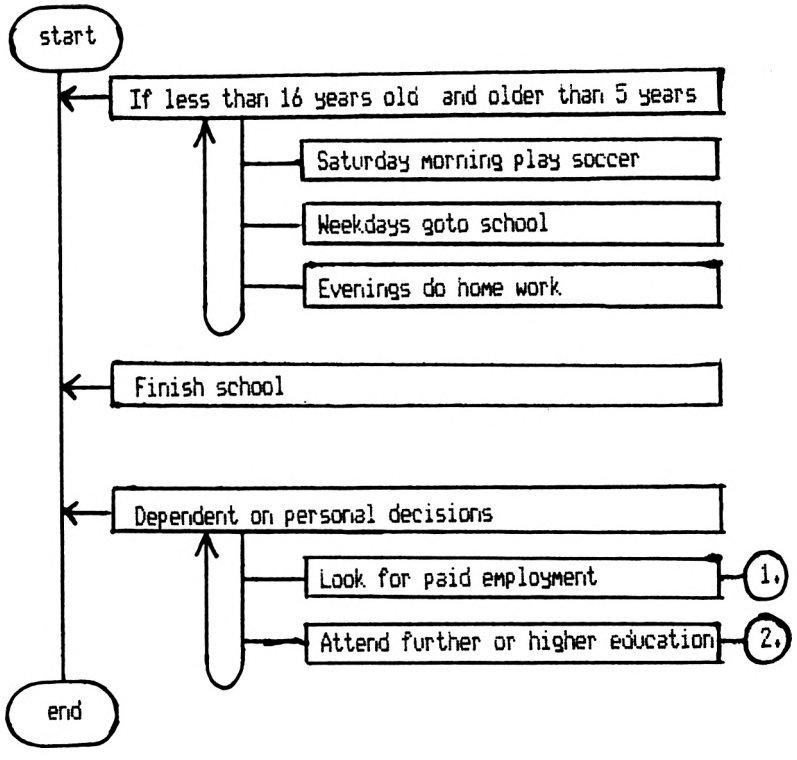


Figure 8.5 By adding a third dimension of repetition, the ability to produce a detailed and realistic analysis of operations becomes a simple logistical exercise

Flow of control

We are now at the stage where we need to bring together the diagram approach and apply it to the realities of keying the program into the CPC 664/464. In terms of the diagram the flow of control relates to the vertical and horizontal lines that are travelled when a specific task is successfully achieved. When dealing with a computer program, as mentioned in Chapter 2, the flow of control is governed by the main control section at the head of the program listing, terminated by the **END** command. This section determines in which order and whether each specific routine is executed.

Using blocks of program—subroutines

By identifying each and every task/subroutine, then drawing up a diagram of its structure, coupled with a diagram of the main control section of the program, the task of creating the actual program in the BASIC computer language is reduced to an exercise in using the features available with the particular language of the computer you are using.

The beauty of a program application such as the telephone directory is that each facility available to the program user fits neatly into a subroutine, with the addition of subroutines for the option menu and elements such as the 'Press any key' routine. Take some time now to consider the creation of diagrams that, if followed by a program user would fulfil the stated aim. Make sure you can do this at both a global level and at a program language level. If you are short on ideas take

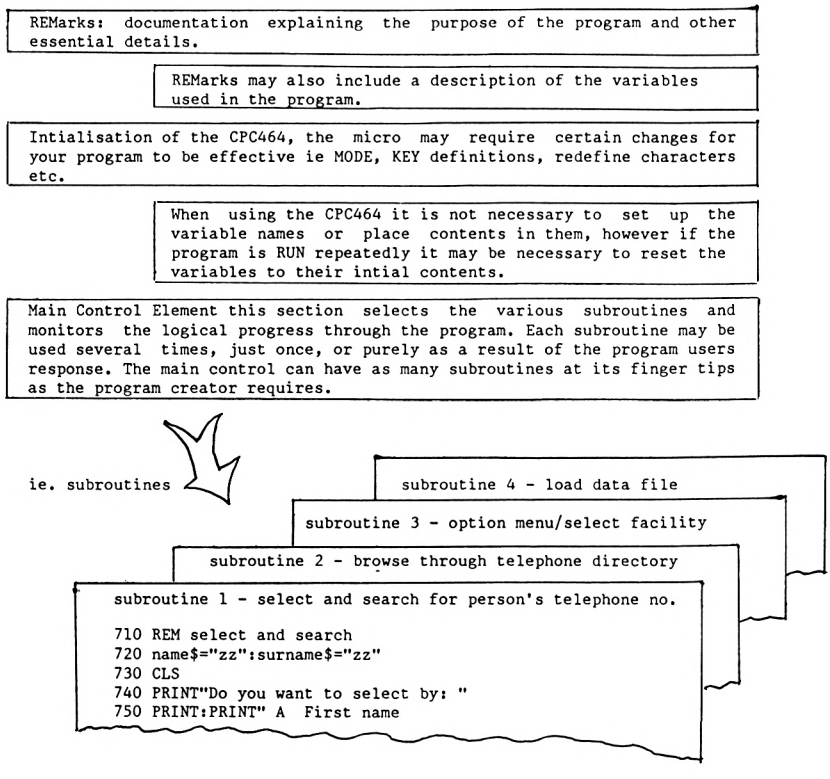


Figure 8.6 The blueprint for a structured computer program

sections of part three of the telephone directory and draw up diagrams in respect of their functions.

A blueprint for a structured program

Look at the program listing in Chapter 2. You should be able to identify each of the sections portrayed in Figure 8.6. Use it as suggested a blueprint for developing first a series of structured diagrams to illustrate your program application. Ultimately these should be capable of being transferred into the instruction lines that make up the program, containing all the facilities you require in your application. Finally compare the essential elements of the program and it should resemble the structure illustrated above.

Chapter Nine

Implementing the Plan: 1

Using the telephone directory application as a model, we are now in a situation of clearly understanding the aims and objectives of the application as seen by the program creator when the idea was first formulated. It is very important to be able to visualise the finished product before embarking on the keying in of the program into the computer.

The most logical method of developing this particular suite of programs is by taking each program in the following order:

1) The 'Create a directory' part, including the 'Add to a previously created directory', i.e. part two of the application.
2) The 'Use the directory' part, the facilities available once the program application is in full use, i.e. part three of the application.
3) The option menu, to join the two main parts of the program together and produce a title page for the application.

This forms a logical sequence for approaching the writing of the programs.

Creating a directory: part two of the application

What operations will the main control element of the program have to perform? To answer this question, let us put into practice the top-down approach diagrams discussed in the previous chapter.

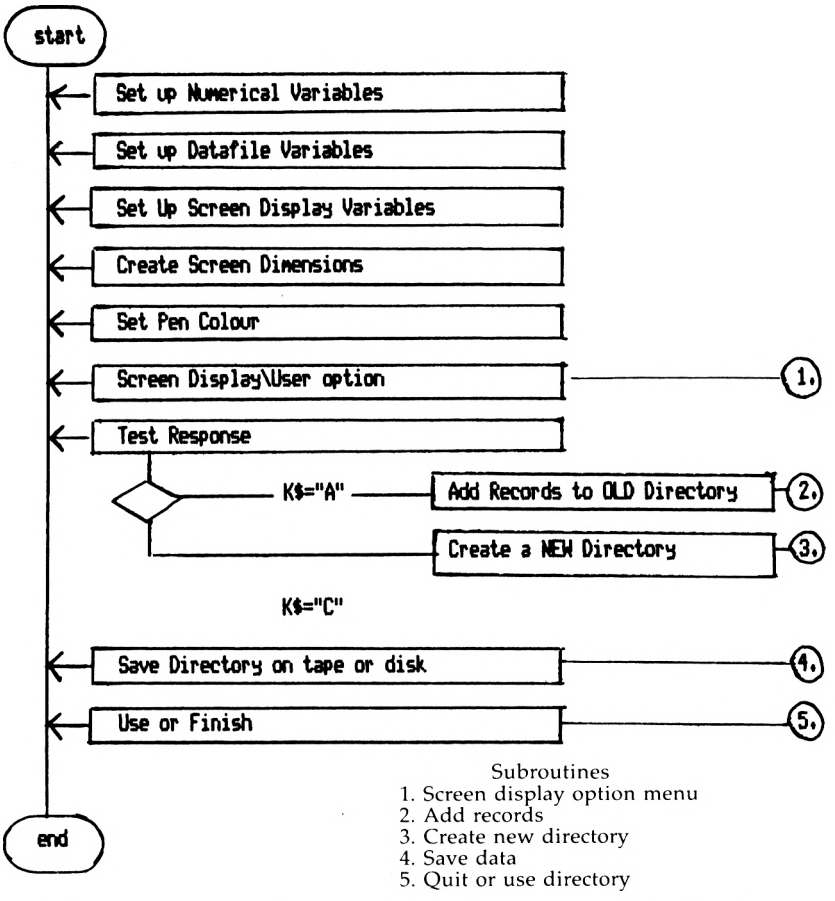


Figure 9.1 Create main control element—part two of the directory

The diagram in Figure 9.1 should now be self-explanatory. The encircled numbers relate to the subroutines which are illustrated in Figures 9.2 to 9.5. Each subroutine performs a neatly structured and virtually self-contained task which, when added to the whole program, works like a cog in a much larger machine. The relationship between the main control block of program and the subroutines can be seen very much in terms of the subroutines being a pack of cards and the main control block being a card player using his cards cunningly to obtain a winning trick.

Link the diagrams in Figures 9.1 to 9.6 to the program listings illustrated in Chapter 2. Remember the diagrams are

the first stage of creating the program, from which the program listing can be surmised.

The main control block of program, as mentioned previously, is the driving house of the program. Thus it can become merely a list or sequence of calling subroutines into action in an order which will ensure the desired end result, i.e. the stated objective of the program. However, in the example we are considering there are two routes a user can take by selecting option `A` or `C` during the execution of the first subroutine. As a result the program places the relevant letter as contents of the variable which has been named/labelled `k$`.

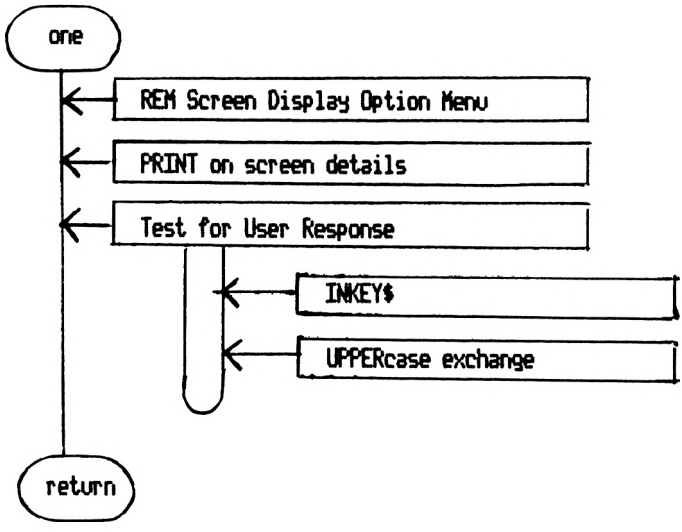


Figure 9.2 Subroutine 1: Screen display and user option

The program control then reverts to the main control element once more, where either the second or third routine is called into action as a result of the user's choice of the options available. On completion of either of these subroutines control once more reverts to the main control block of program.

A subtle programming technique is employed to cut down on the time spent keying in the program instructions. Notice that subroutine two merely loads the previously created directory from data cassette or disk, setting the relevant variables that control the number of records stored etc. to hold the appropriate contents. Changing the contents of `k$` to the letter `C` results in the automatic execution of the third

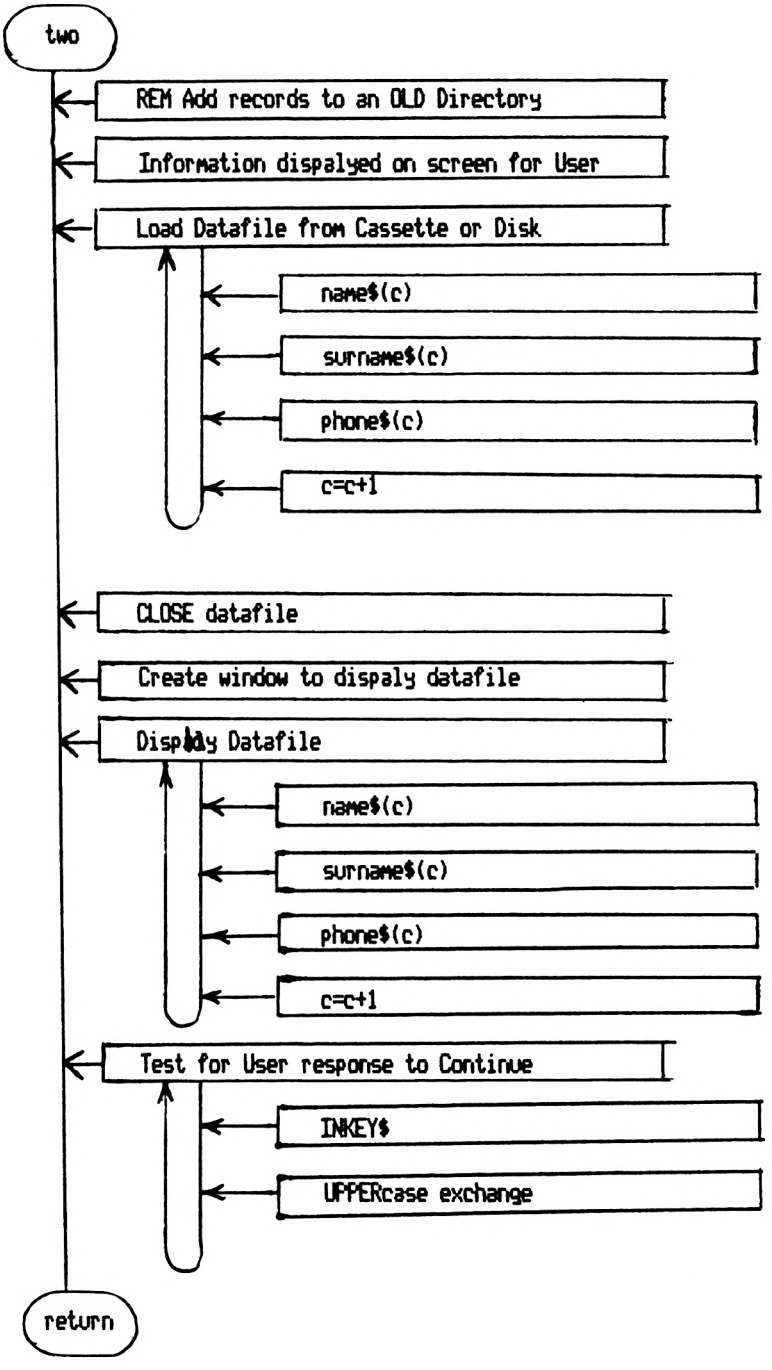


Figure 9.3 Subroutine 2: Load and display `OLD` directory

subroutine because once the task of loading the previously created directory has been performed, the program is written so that both adding and creating an entirely new directory use the same program instructions, i.e. subroutine three. On

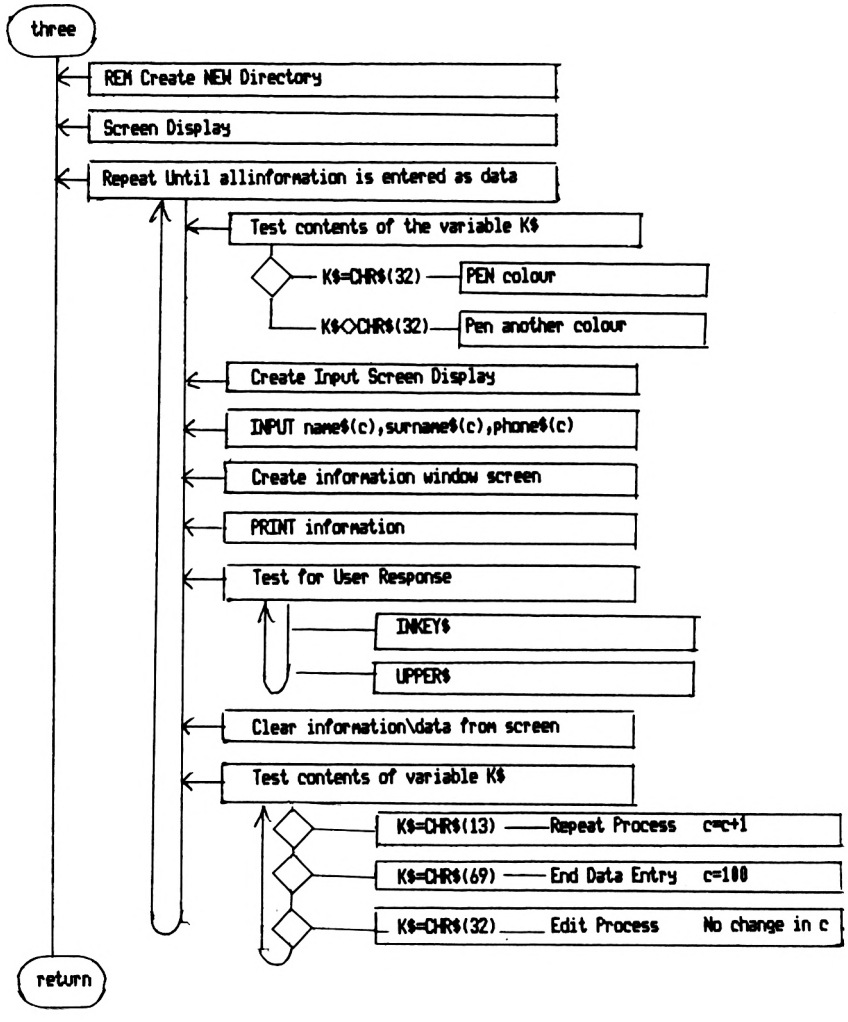


Figure 9.4 Subroutine 3: Create directory data entry

completion the new or added-to directory is saved onto cassette or disk (subroutine four) and the user given the option to finish or to load the next program (part three) and use the directory immediately, i.e. subroutine five.

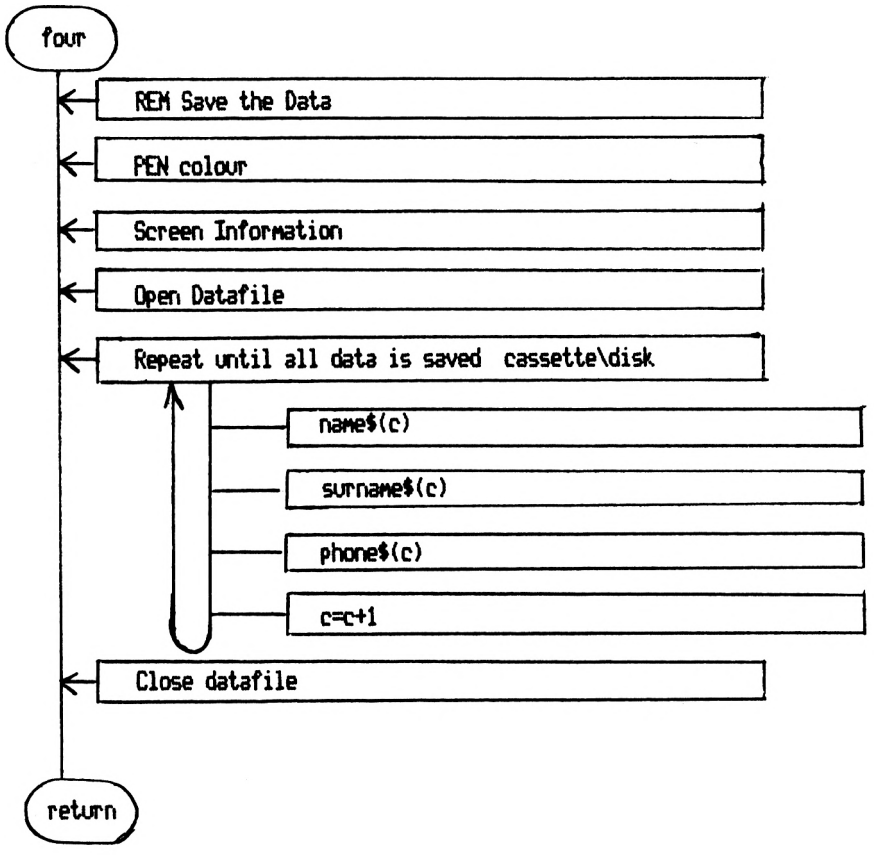


Figure 9.5 Subroutine 4: Save data previously entered on cassette or disk

Notes on the subroutines of part two

Each diagram clearly relates to a specific task of the application and should be easily readable by following each operation across the page. The sequence of operations should follow down the page.

The first two subroutines provide simple sequences of events, with a number of repetitive operations which are either terminated by the program user making the appropriate response or by displaying the data i.e. names and phone numbers on the screen or loading from the storage system to the computer.

The repetitive loop, subroutine one, is a simple loop that

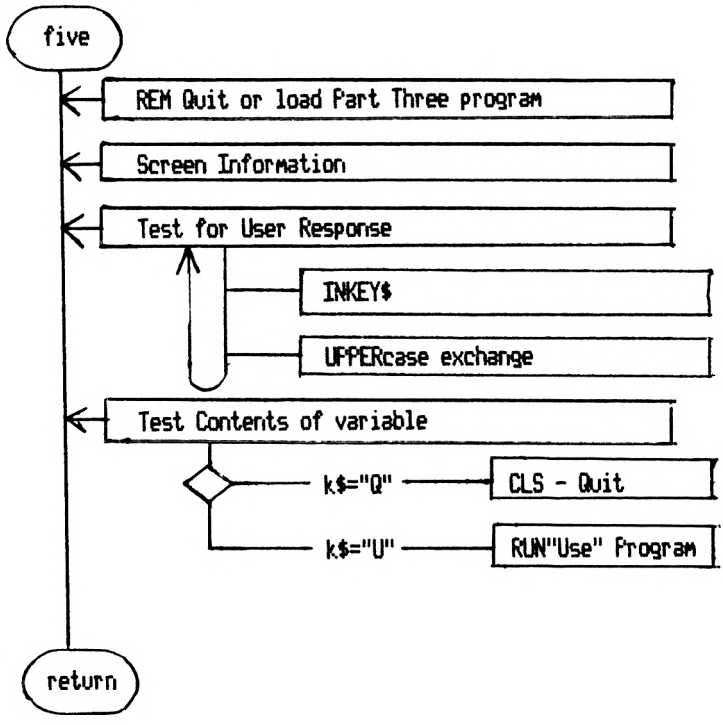


Figure 9.6 Subroutine 5: Quit or load 'Use' program

allows the program to wait on the **INKEY$** statement until the appropriate key is pressed, i.e. a choice is made by the user. BASIC command words within a diagram should be used sparingly but effectively to demonstrate a particular operation that is to be carried out. This is why each **INKEY$** is generally followed by the command word **UPPER$**.

Make your diagrams an intermediate step between the screen display sketches and the program lines themselves. Various lines of instructions should be identifiable, but embedded in phrases and words that clearly explain the operations to be carried out.

Variable names, as in subroutine one, again should be used sparingly, but use them to demonstrate as in this case that **DIM**ensional array type variables are repeatedly saved or loaded from cassette or disk or printed on the screen until the data held within that variable's dimension is used. For example, the second repetitive loop, subroutine two, prints every name and phone number held in memory on the screen.

Subroutine three sees a further complication of the repetitive loop of operations. The subroutine is initialised by screen display and all the remaining operations are concerned with the adding of names and phone numbers. The two decision sections relate to the correctness of the record or the termination of the directory creation. The actual **INPUT** of data is simply a sequence of operations which will be coupled with various screen layout procedures when put into program instructions.

Throughout a planning stage, relate to your screen display sketches and visualise the need to alter colours. Whether this is done by physically changing the ink or changing the text cursor being used is really up to the individual likes and dislikes of the programmer.

If you find it difficult to jump straight into the drawing up of diagrams which include all the repetitive loops and the decisions to be made, begin by listing the operations in a preliminary diagram, a simple sequence of operations. Add to this by making a second diagram, filling in some of the detail. Which sequence of tasks will be repeated until a condition is met? At which points will a 'Press any key' routine have to be included? When will the computer be processing under its own steam, such as loading data from a cassette or disk? Such computer processing is generally assured by means of the program instructions having a loop built into the program at that particular point.

Chapter Ten

Implementing the plan: 2

By using part three of the telephone directory further complications of programming and various techniques of signalling operations can be examined. Briefly, two of the most useful programming methods are: firstly, the creation of a program loop consisting of several operations, any of which

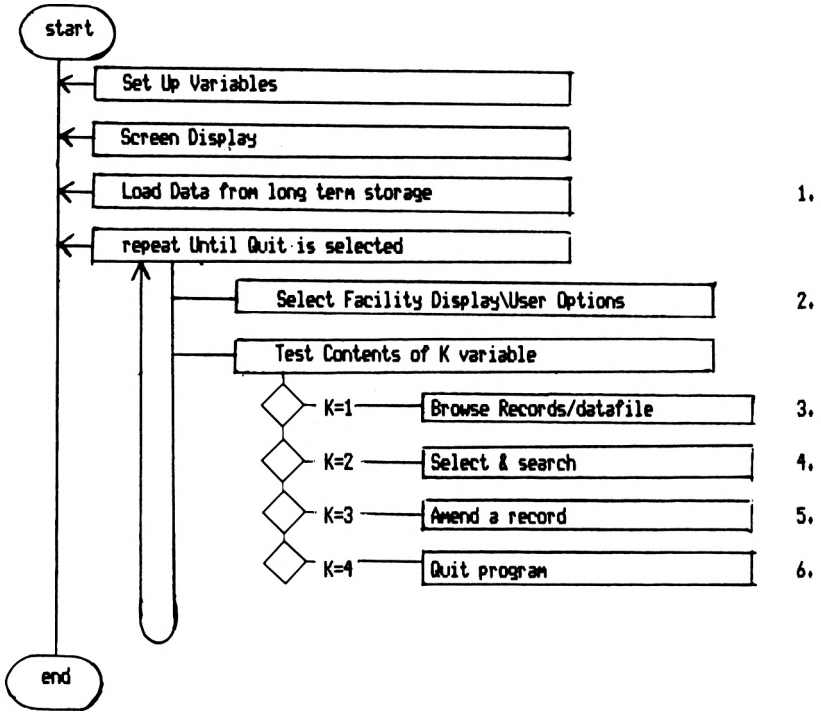


Figure 10.1 Main control element of 'Use' program—part three of the directory

the user can choose as and when desired; and secondly, the execution of an operation if and only if a certain other event has taken place.

Using the directory: part three of the application

Figure 10.1 illustrates the main control element of the program. The remaining figures illustrate the top-down diagrams of the subroutines called by the main control element. Once the variables have been created, and the screen set, the telephone directory data is loaded from tape or disk by a subroutine called by the main control element. Once this has been completed a repetitive loop is entered into by the program user. The only exit from this loop of program instructions is by the user choosing the 'Quit' option, i.e. the important variable k having a contents of 4. The relevant program lines are:

```
110 WHILE k<>4
120 GOSUB 380    :REM select facility
130 ON k GOSUB 550,710,950,1230
140 WEND
```

While the program is executing the 'Select facility' subroutine (program lines 380 to 540 of part three) the program user chooses a number between 1 and 4 inclusive. This numerical choice becomes the contents of the variable k. The command ON k GOSUB is a special and very useful BASIC instruction line. The GOSUB instruction can be followed by any number of subroutines that the program user may require to use. If the variable has as its contents '1', the first subroutine is executed, if '2' the second subroutine is executed and so on. The capability of this instruction line is particularly useful for the function of reacting to a menu displayed on the screen. If the variable content is not equivalent to any of the listed subroutine positions then *no* subroutine is executed; it is therefore necessary to nest the ON GOSUB line within the loop to guarantee the program will execute only the facilities it is asked to. The choice of options is therefore immediately validated. Each facility will be executed; on its completion the control is returned to the program loop enabling a further

option to be made. If an incorrect key is hit the loop ensures the program does not end but allows another choice to be made.

Let us now consider each of the subroutines as they appear in the program listing. The first subroutine called into action by the main control element loads the data which will provide the telephone directory information specific to each user.

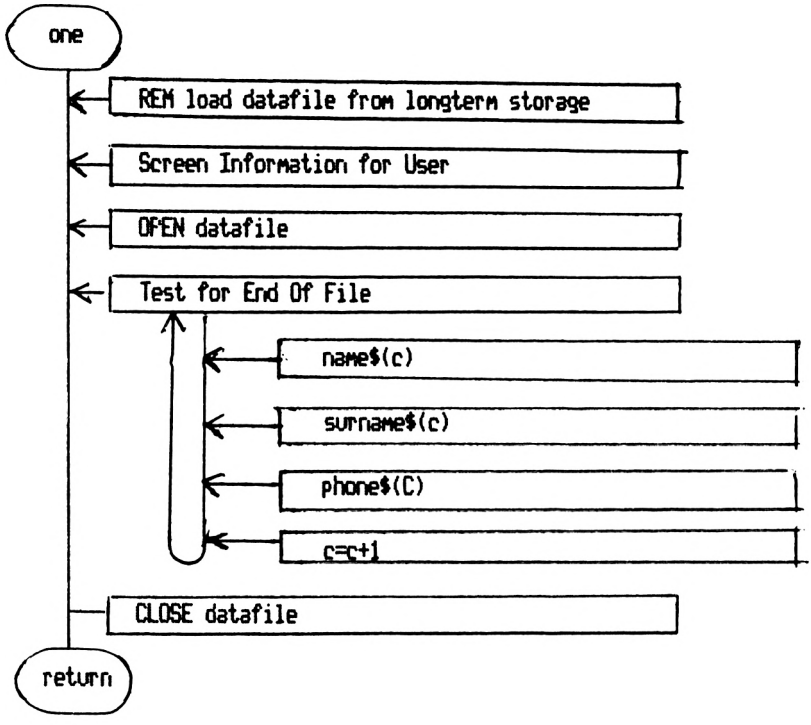


Figure 10.2 Subroutine 1: Load data file from cassette or disk

Once again every operation that the computer executes from the programmed instructions follows a set procedure. First the screen is cleared and information to the user is supplied. The subroutine specific program lines are now executed. Because the data is to be loaded from a peripheral device, i.e. tape or disk, a special data channel has to be opened. Once communication is enabled the computer begins the repetitive procedure of loading name, surname, and telephone number and placing them in memory until all those from that user's directory are loaded. The loop is then terminated because a

code is sent from the disk or tape to signal the end of the data file, i.e. **EOF** is the BASIC instruction. At this point in the program the counting variable's contents is noted and remembered by placing the same contents in the variable, **record**. In order to close down the communication link the program executes the sequence operation of **CLOSEIN** once the loop has been terminated. The subroutine is now complete and the program flow returns to the main control element with the advantage of now having the user's telephone directory stored in its memory.

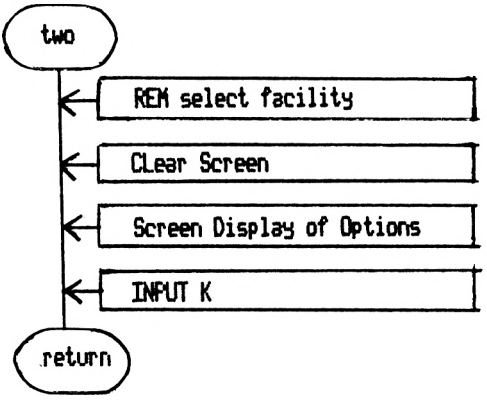


Figure 10.3 Subroutine 2: Select facility and user response

Before the user can actually make a choice, he or she has to be given the appropriate information. The 'Select facility' subroutine provides this information and allows the user to make the choice of facility to be executed. It is simply a sequence of screen display instructions that are executed in a logical order before returning to the main control element once more. The screen is cleared, the information is **PRINT**ed on the screen and the program instruction **INPUT** waits for the user to hit a key and press the <**ENTER**> key. This subroutine will be repeated on numerous occasions as it is called from within a repetitive loop in the main control element of the program. If an incorrect choice is made the subroutine will be repeated without another subroutine being executed first; if a correct choice is made, i.e. keys 1 to 4, then one of the following subroutines is executed before the 'Select facility' subroutine is once again performed.

The program facilities subroutines.

Browsing through the telephone directory

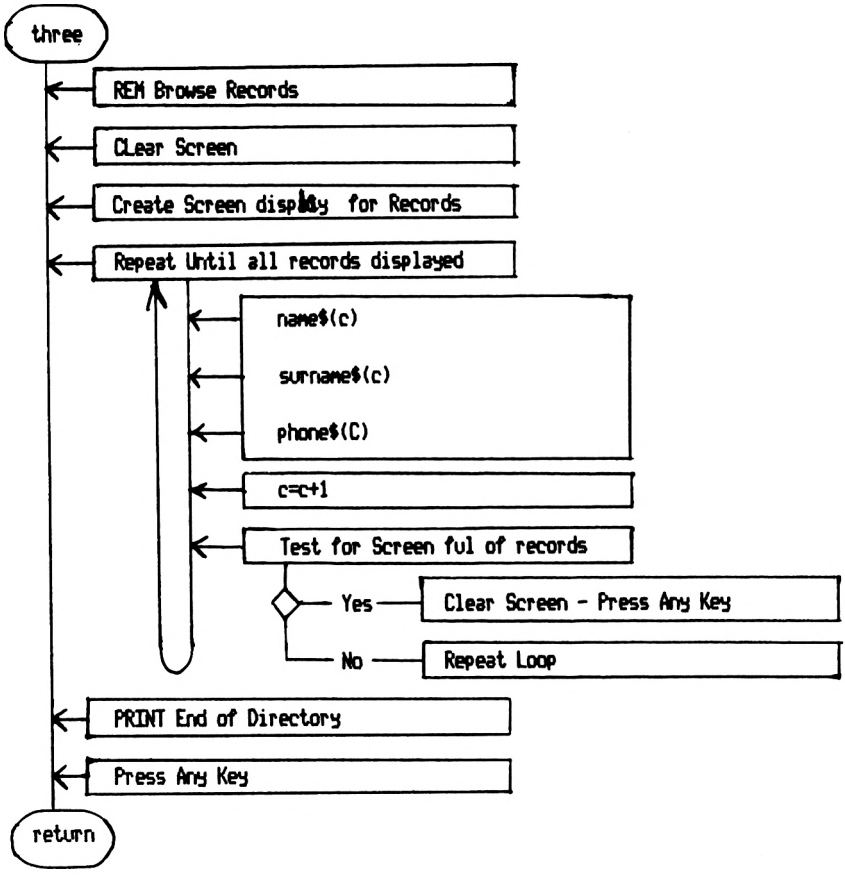


Figure 10.4 Subroutine 3: Browse through records

The browse facility prints on the screen all the names and telephone numbers entered by the user. They appear in an order known as key order, i.e. the order in which they were entered while creating the directory.

If the screen were long enough to accommodate all the directory on one screenful of information this subroutine could be merely a sequence of operations with the repetitive print of the directory's information. This is not possible so the program has to perform a checking operation to ensure that

only a full screen of information is displayed at any one time. To continue on to the next screen of information the 'Press any key' routine has to be executed, as in Figure 10.8.

The logical order once again is: clear the screen, set the screen display, repeatedly print the names and numbers on the screen; meanwhile check that the screen is full of information or not, execute the appropriate operation. On completion of the repetitive loop an 'End of directory' sign is displayed and before the subroutine is terminated and control returned to the main control element, the 'Press any key' subroutine is called and executed.

Select and search subroutine

Again the subroutine follows a standard format of screen display instructions waiting for the user to respond and then processing the information.

The screen is cleared, information to assist the user in his or her response is printed on the screen. A choice of key to press is given. As a result the user is then asked for either surname or first name, determined by the initial choice. The name by which they want to search the directory by is then placed as contents in one of two variables, i.e. `name$` or `surname$`. Note that both are similar to the variables we have referred to directory data by, but without the trailing brackets containing the usual variable (see Chapter 14). Once the name/surname is entered, further screen display is added and the computer processes whether the entered name/surname is the same as any of those stored within the data of the directory. This is achieved once again by comparing each of the names in the directory with the entered name one at a time from within a repetitive loop. If the names compared are found to be the same the `IF—THEN` statement is implemented and the flag variable contents is changed to '1'. This means that the instruction 'No record found' will not be displayed on the screen. The second part of the `IF—THEN` statement causes the computer to print the related directory information on the screen. This process is repeated by the use of a `WHILE—WEND` loop combined with the counting variable `c` and the comparison of whether its contents is same as the number of records stored as contents in the variable `record`.

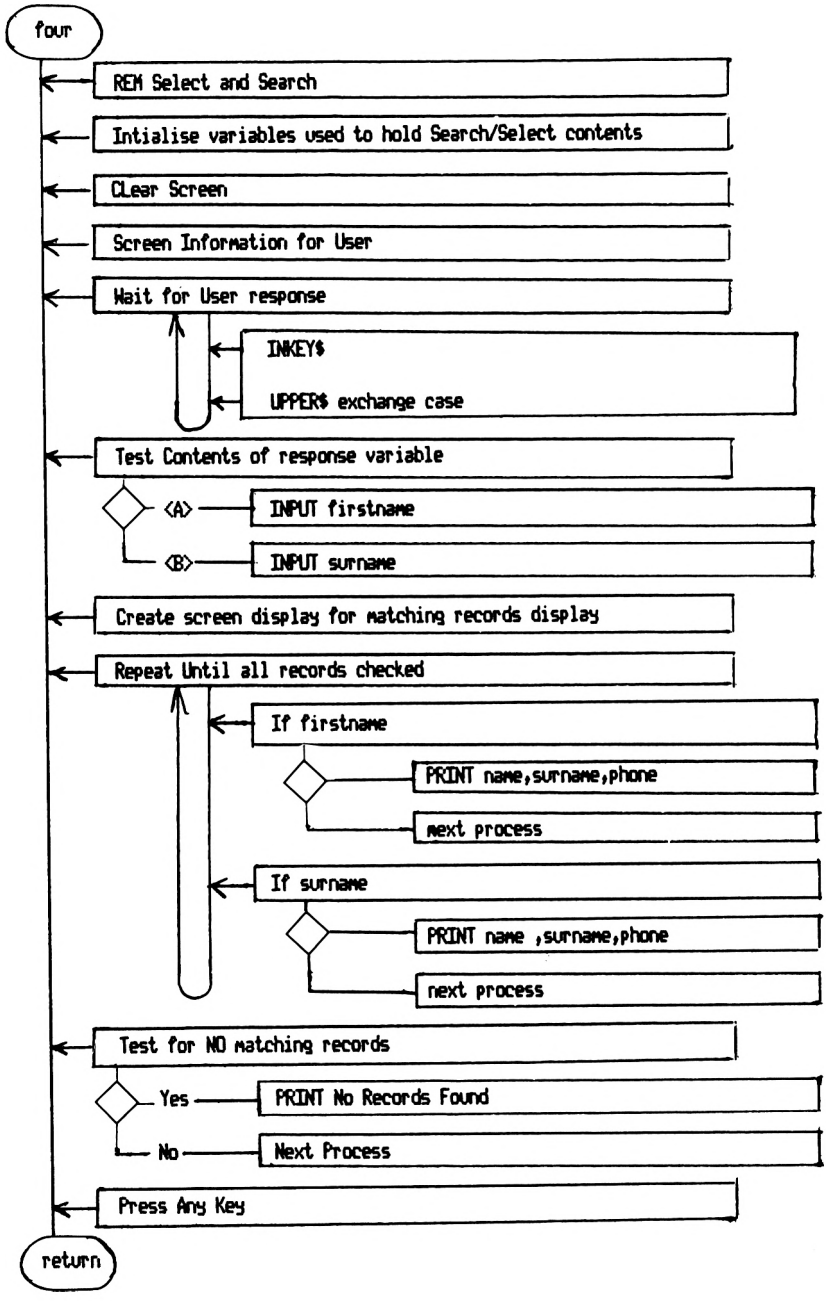


Figure 10.5 Select and search for matching records

Once all the records have been tested/compared the loop is terminated. A test is made to see whether any records were found to be the same as the name/surname asked for, the use of the flag variable `f`. In order to terminate the subroutine once more the use of the 'Press any key' routine is implemented. On the user responding to the 'Press any key' instruction the program returns control to the main control element of the program.

Amend a record subroutine

The screen is prepared once more for action. The job to be done is explained by a simple screen instruction. It is impossible to amend a record without first having found it, so what would be easier than actually repeating the execution of the select and search subroutine? To avoid printing exactly the same instructions and to make them more relevant to the situation the subroutine is begun at a line number later in the block of program. The name is found or not found as the case may be. The control of the program returns to the 'Amend' subroutine from the 'Select' subroutine. If no records are found the subroutine might as well finish, as you cannot change a record that is not in the directory anyway. To continue let us assume the record has been found. Each record found has a number allocated to it and the program instruction asks the user to `INPUT` the number of the record they wish to amend. The record is amended by executing a series of sequence type operations.

This record so far has only been amended while stored in the computer's memory. In order to save it for long term use the computer has to save it onto either tape or disk. The operation is performed by saving the whole directory once more and this is very time consuming, compared to only adjusting the single record. To cover all the various forms of file handling is unfortunately outside the scope of this book, but the planning and structured design is the same.

The operations involved in saving data are: open communication link, screen instructions, repetitive loop to save each and every name, surname, and telephone number—if there are twenty records on the directory this loop will be

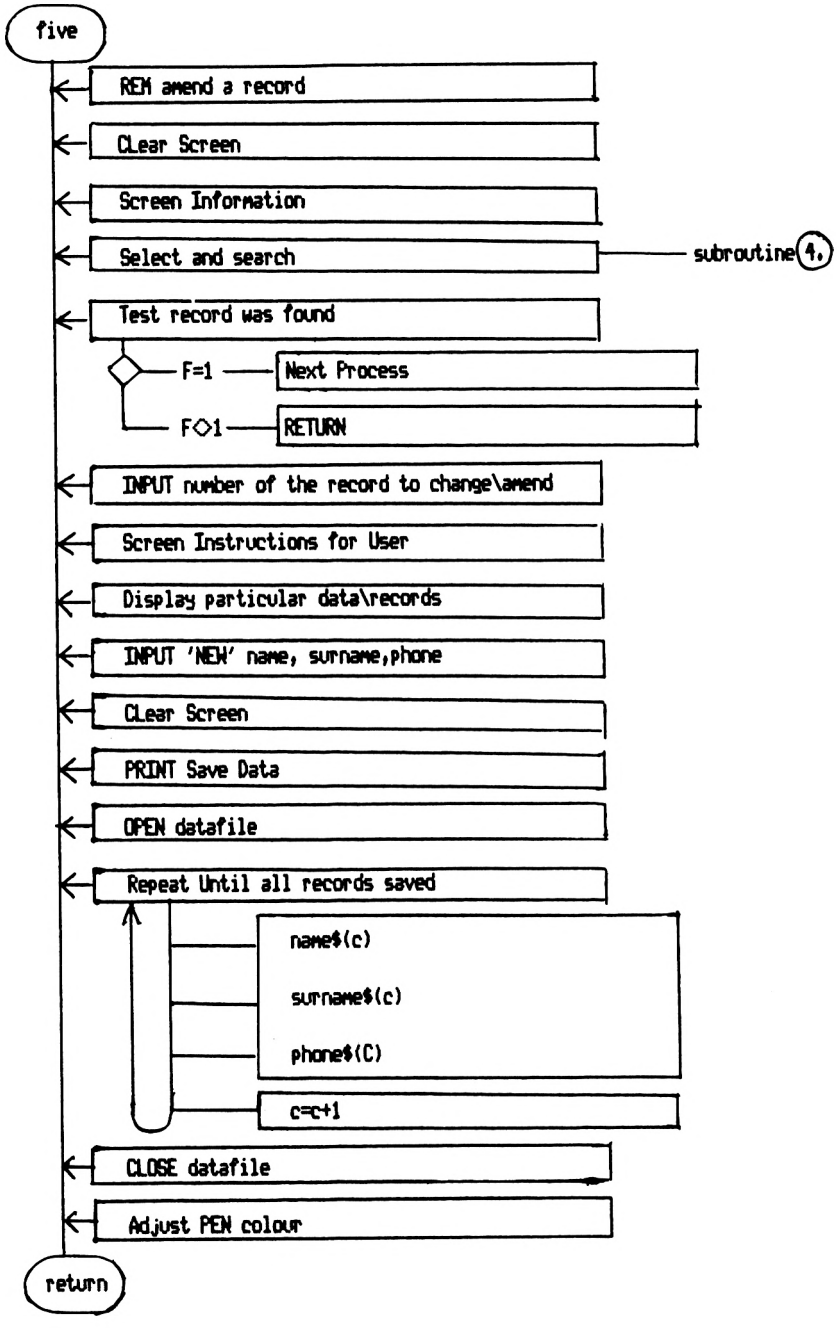


Figure 10.6 Subroutine 5: Amend a record

repeated twenty times—, close communication link, return program control to the main control element of the program.

Quit subroutine

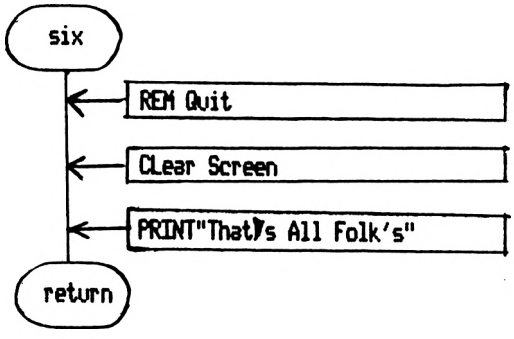


Figure 10.7 Subroutine 6: Quit

This is a straightforward sequence of events: information is printed on the screen and then control returned to the main control element which terminates the whole program because the variable k has a contents of 4. The screen is left displaying the message 'That's All Folks'.

Press and key subroutine

This subroutine can operate in conjunction with any other as it merely uses the bottom line of the screen. The program lines wait for the user to respond indefinitely, whereupon the subroutine is terminated and control is returned to the program area that is called the 'Press any key' subroutine.

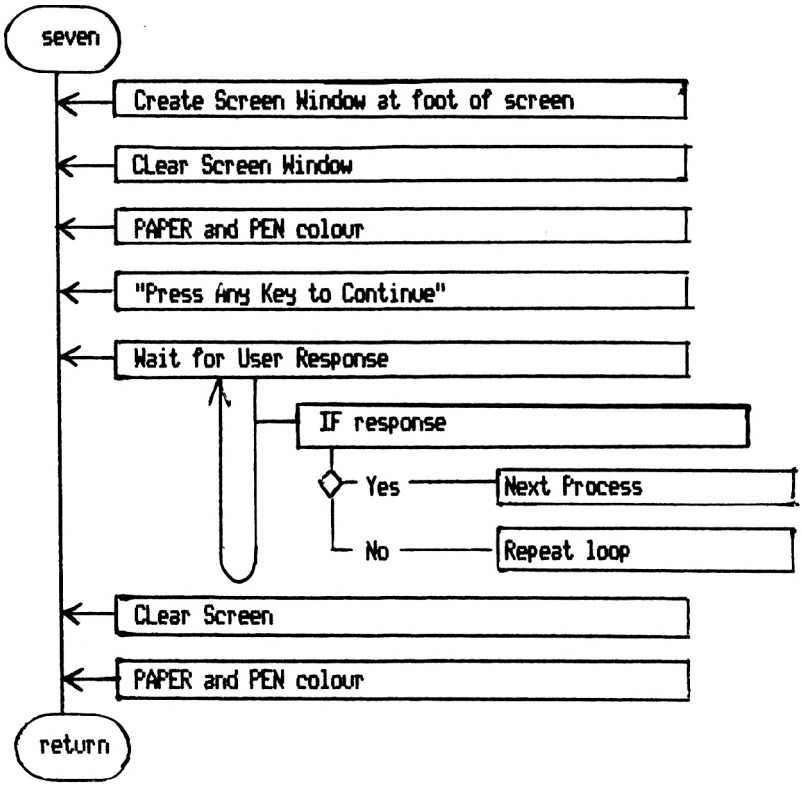


Figure 10.8 Subroutine 7: Press any key subroutine

The option menu: part one of the application

For completeness Figure 10.9 illustrates the option menu which enables the user to either create or use the telephone directory. Examine it and try to talk your way through it. Then compare it to the program listing, Chapter 2, or the program as you have it on tape or disk.

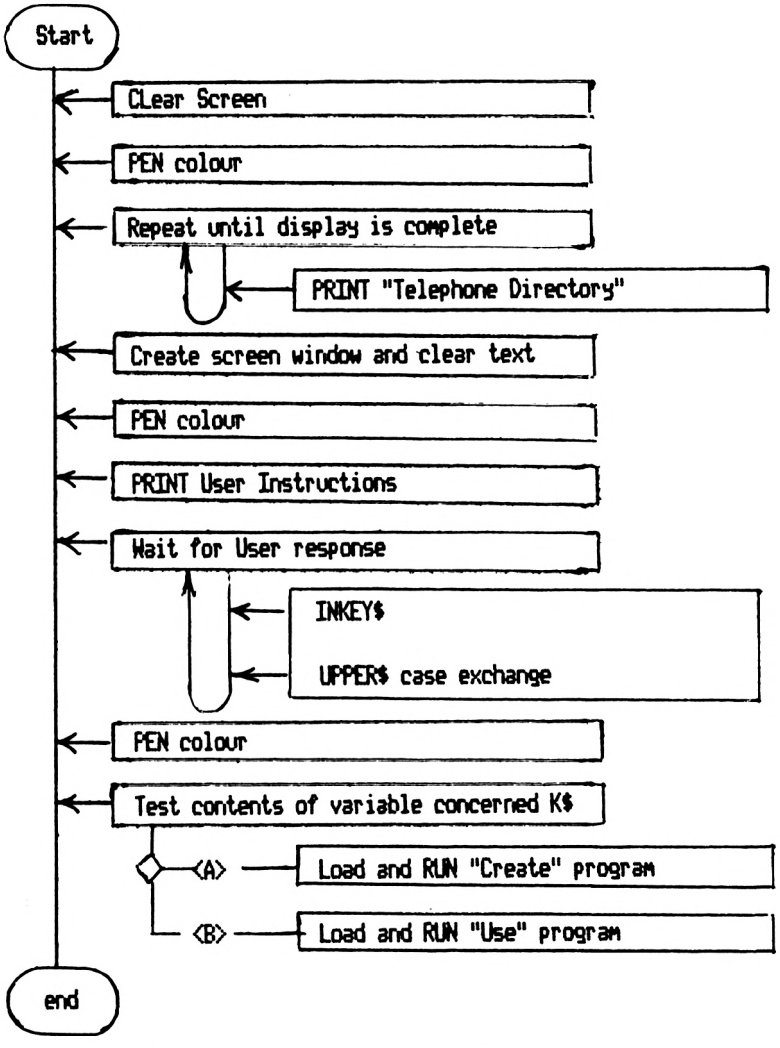


Figure 10.9 Program structure for part one of the telephone directory

Section E
Handling Text—The Key to Information Storage

Chapter Eleven

Strings, string variables and the $ symbol

A string of characters can be one or more alphabetical, numerical, or other keyboard symbols presented together. Using direct command mode we can place on the screen a string of characters thus: `PRINT "6ab*G0 87"` <**ENTER**>. Similarly a readable sentence is also considered in terms of a string of characters, thus: `PRINT" This sentence is a string of characters"`. Note that a blank space counts as a character within a string. This is an important fact when considering that when entering data into a database, such as the telephone directory, each piece of information is actually placed as contents in a string variable. Therefore if the user is to place an additional blank space at the end of, say, the surname, the computer will read that space into the string variable along with the letters that make up the surname. Try this with the telephone directory. Then try to select that particular surname, but leave out the blank space—you will find the program will not find the particular surname you entered previously with the blank space on the end of it.

The contents of a string variable

The **$** symbol when suffixed to a variable, enables characters to be placed as contents within that variable. A simple manipulation of these strings might be as shown in the lines 10 to 90 below. By using the + symbol, strings of characters can be lumped together to make a longer string of characters, and placed as contents in a new variable, labelled with a different name.

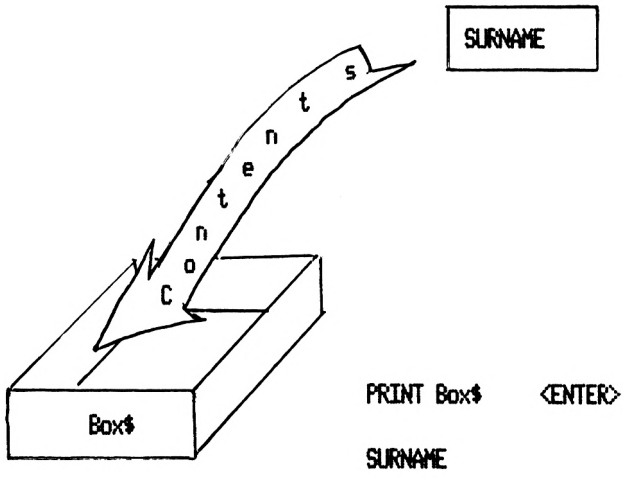


Figure 11.1 The contents of a string variable

```
10 INPUT"A firstname is";A$
20 INPUT"A surname is";B$
30 Z$=A$+B$
40 Q$=A$+" "+B$
50 Y$=A$+CHR$(32)+B$
55 PRINT
60 PRINT Z$
70 PRINT
80 PRINT Q$;" This is Q$"
90 PRINT Y$;" This is Y$"
```

The functions CHR$ and ASC

In the above program lines 40 and 50 the same result has been achieved by different means. The second method, line 50, is probably just a neater, more professional, more standardised method. The command `PRINT CHR$(13)` has the same effect as pressing the <**ENTER**> key. The function `CHR$` is an abbreviation for character code. Each character has a code number referred to as an ASCII code. When ever possible use the form of `CHR$` (ASCII for the keyboard character) instead of the character in brackets for `PRINT`, `LET`, `IF—THEN` and `WHILE` conditions. The computer's processing speed will be enhanced and the program as a whole will be more professional in its execution. Unfortunately it will mean that to read

through the program listing, it will be necessary to refer to a table of ASCII codes.

The function ASC has the inverse effect, e.g. **PRINT ASC("E")** <**ENTER**> will display on the screen the number 69. The use of this function is a little more specialised. It also can be used as part of the conditions set up in a **WHILE** loop and an **IF—THEN** statement.

Calculating the number of characters in a string

The length of a string can be determined by the BASIC command word **LEN** with the particular string variable concerned in brackets after it:

```
10 INPUT "A surname is";B$
20 length=LEN(B$)
30 PRINT"The surname ";B$;" has "; length;" letters in it."
```

The immediate use for this command in terms of a database is during the set-up and creation stage. As part of the set-up procedure the program can ask the user to define each field of information in terms of the title of the field and the maximum length of the information to be placed in that particular field of information. The advantage of this type of program structure is that it simplifies designing the screen layouts for data entry and browsing of data while using the database. The function **LEN** would be used to test the length of each string as it was entered at an **INPUT** stage. If it is less than the previously determined length it would then be stored as an array variable in the database. The structure would be:

```
10 c=1
20 INPUT "Number of names";number
30 INPUT"The max. letters in the names";length
40 DIM Name$(number)
50 WHILE c<number+1
60 PRINT c;
70 INPUT" Name";enter$
80 IF LEN(enter$)>length THEN 110
90 Name$(c)=enter$
100 c=c+1:GOTO 120
110 PRINT "Name toolong"
120 WEND
```

The use of UPPER$ and LOWER$

Both are very useful functions but as a programmer one generally has a preference and uses it to the exclusion of the other. The function they perform is to adjust a string variable contents of alphabetical characters to the form stipulated. Thus `UPPER$` will convert all letters to the upper case form and `LOWER$` will convert to the lower case form. The essential advantage in any program is that a program then only has to validate a user's response to one case. The program listing also looks neater and more professional. In terms of the program user it then does not matter whether the computer has its **caps lock** facility on or off.

STRING$ and SPACE$ used for short cuts

To make a program more effective it is a great advantage to make full use of the facilities the computer provides. `STRING$` is one such facility. Its format is `STRING$`(a whole number,"characters ").

```
10 T$ = STRING$ (5, "repeat")

20 PRINT T$

RUN
```

Figure 11.6

The same task can of course be fulfilled by the program lines in Figure 11.7.

```
10 FOR C = 1 TO 5

20 PRINT "repeat";

30 NEXT

RUN
```

Figure 11.7

A further development might be to replace the characters in the **STRING$** format with the function **CHR$**.

```
PRINT STRING$ (5, CHR$(69))
```

Figure 11.8

The function **SPACE$** is a special form of the **STRING$** function which produces a given number of blank spaces, for example, **PRINT SPACE$(5)** will produce five blank spaces. The use is for screen layouts, creating a **TAB** facility in a 'word processing' program.

As your programming ability and range of knowledge develops more functions will become available to you as tools assisting the ease with which you will be able to perform program tasks. My advice is to take it slowly and get used to using the fundamental commands. When you are faced with a seemingly insoluble problem, consult the BASIC keyword reference of the computer; with a little imagination the function will be there to complete the task you want the computer to perform. A function can generally be broken down into more fundamental program lines. As you become confident browse through the BASIC keyword reference provided with the computer to arm yourself with these tools.

Chapter Twelve

Manipulating strings to your advantage

The magic of any computer program that is functional is its ability to reliably and repeatedly carry out essentially boring and repetitive tasks time and time again without becoming inaccurate. For example, look for a person who has an address of '1 Dodd Street' from a directory of 20 000 people. A well designed program could come up with this answer within a matter of seconds. The most commonly used procedures are those of searching and sorting, and the CPC 664 and 464 both offer various facilities to search and sort. These facilities are effectively determined by the program user but the options available have to be written into the program first.

The essence of searching for a particular name, or in computer terms a string of characters, is by using the format `IF a$=b$ THEN found$=a$:PRINT found$`. This testing procedure is set within a loop such that every variable in the list is tested against the required variable. When the two strings of characters match they are printed on the screen. Just such an example is used in the telephone directory suite:

```
710 REM select and search
720 name$="zz":surname$="zz":f=0
730 CLS
740 PRINT"Do you want to select by: "
750 PRINT:PRINT"<A> First name
760 PRINT:PRINT"<B> Surname"
770 WHILE k$<>"A" AND k$<>"B"
780 k$=INKEY$
790 k$=UPPER$(k$)
800 WEND
810 PRINT
820 IF k$="A" THEN INPUT"Which first name";name$
```

```
830 IF k$="B" THEN INPUT"Which surname";surname$
840 PRINT:PRINT "   ";f$,s$,ph$
850 WINDOW 1,40,10,21
860 c=0
870 WHILE c<record
880 IF k$="B" THEN 900
890 IF name$=name$(c) THEN f=1: PRINT c;name$,surname$(c),phone$(c)
900 IF surname$=surname$(c) THEN f=1: PRINT c;name$(c),surname$,phone$(c)
910 c=c+1
920 WEND
925 IF f<>1 THEN LOCATE 10,8:PRINT " No Record Found "
930 GOSUB 1280
940 RETURN
```

The essence of sorting is probably a little more complicated. It centres around the ability to decide whether a string of characters is 'less than' or 'greater than' another string. 'Less than' translates to being alphabetically nearer to the letter 'A' and 'greater than' translates to being nearer to the letter 'Z'.

The method the computer uses is to convert each character to its ASCII number equivalent. If the string is of multiple characters it will compare each character individually. Therefore it is logical to say that the ASCII code in decimals for the letter A is 65 and for Z is 90. The problem is that for lower case letters the ASCII decimal codes range from 97 for 'a' and 123 for 'z'. Therefore if you require the sorting of either case to make no difference, ensure that during the program routine you use the UPPER$ or LOWER$ facility available on the CPC 664/464.

A very untidy but workable method of sorting three string variables into alphabetical order is as shown below. It is not validated to accept both upper or lower case letters, so it is clearly not foolproof. However it serves as a useful demonstration of the procedure required to print a list in alphabetical order.

```
2 CLS
5 c=1
8 PRINT "Three words, please"
9 PRINT "Type, separate each one by a comma."
10 INPUT ;a$,b$,c$
12 PRINT:PRINT
15 PRINT "Alphabetical Order is:"
```

```
17 PRINT
20 WHILE c<4
30 IF a$<=b$ AND a$<=c$ THEN temp$=a$:f=1
40 IF b$<=c$ AND b$<=a$ THEN temp$=b$:f=2
50 IF c$<=a$ AND c$<=b$ THEN temp$=c$:f=3
60 PRINT temp$
70 IF f=1 THEN a$="zzzzzzzzzz"
80 IF f=2 THEN b$="zzzzzzzzzz"
85 IF f=3 THEN c$="zzzzzzzzzz"
90 c=c+1
100 WEND
110 PRINT
```

The theory develops around finding the string nearest to the letter 'A' position; that particular string is destroyed or more accurately changed to consist of a series of z's, alphabetically last. In this way the next in alphabetical order can be found, printed on the screen, and is then altered to a series of z's. The third in alphabetical order is then found, and so on.

The problem with this method is its laborious and time-consuming task of writing into the program every single combination for any more than three strings being sorted.

Many different methods of sorting have been devised. Each has its own advantage: some account for upper and lower case variables, some destroy the original variable's contents so that it can be used again, others do not, some are much faster at doing the task than others. All use a series of variable contents manipulations from one variable to another.

The theory is straightforward: each item in the list is repeatedly searched, using item one as its standard until it finds an item in the list which is alphabetically closer to 'A'; that item then becomes the standard to test against. The process is continued until the end of the list is reached. The current standard is printed on the screen. The standard item is now disregarded and converted to the alphabetical position closest to 'Z'. The standard becomes item number one again and the process is repeated in once more. The process continues until all the items have been alphabetically sorted and listed on the screen.

There are other methods that use dimensional arrays and actually alter the position, by changing the variable's subscript number. The contents of the two variables are held in

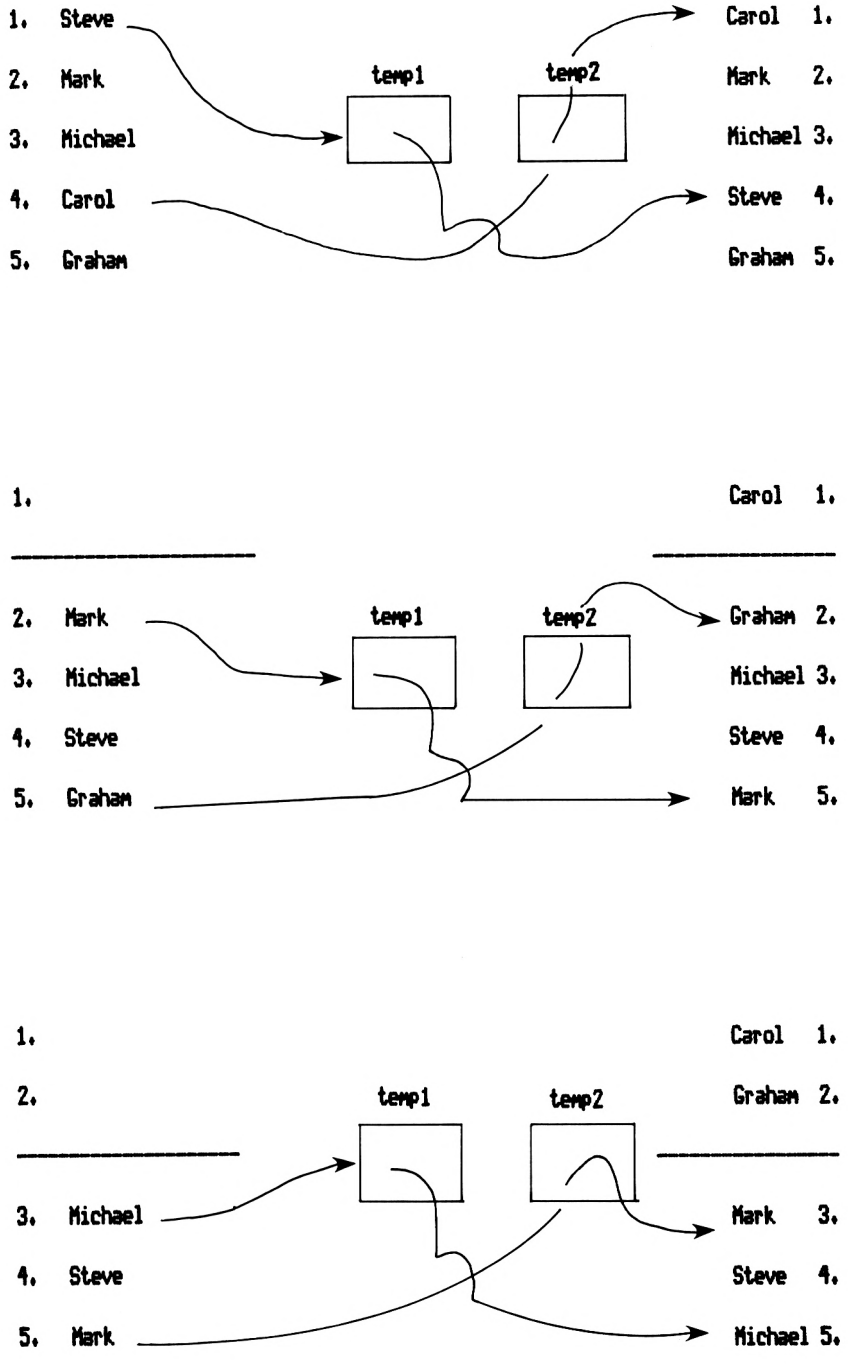


Figure 12.1 Alphabetical sorting procedures—the use of temporary stores

temporary variables and then switched into order as a result. The number of elements compared are then adjusted to make sure that those already placed in order are not compared again. Figure 12.1 visualises the sequence of events. The structured diagram might be something like Figure 12.2, demonstrating another very useful application for creating structured diagrams of a specific task. See if you can create your own methods of searching and saving. Start from the point of scribbling ideas in the form of a sketch diagram.

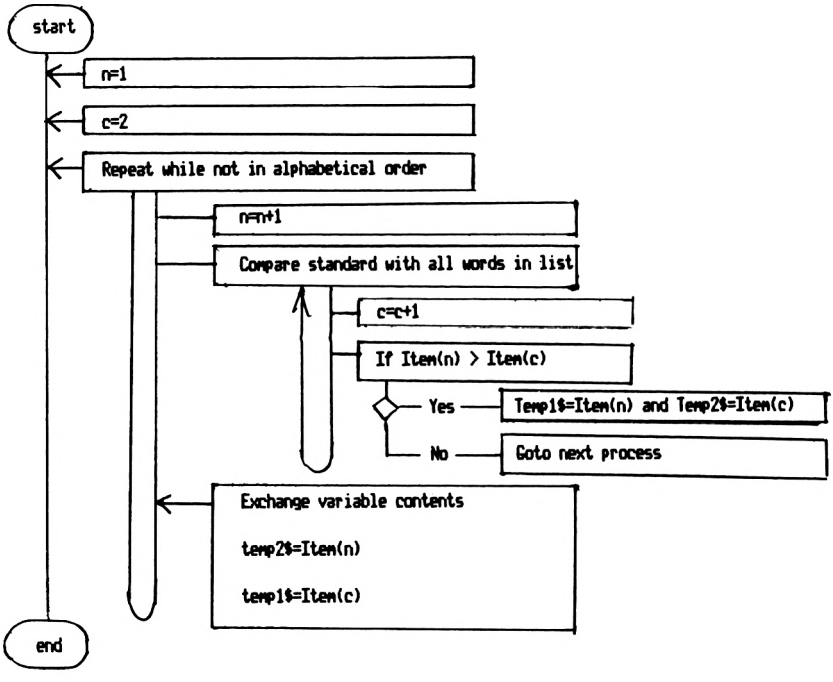


Figure 12.2 Structured diagram illustrating a sorting procedure

CPC 664/464 facilities available for use

There are several functions available that can be introduced into the structure of a program that will enable greater flexibility for the program user to determine the exact part or section of a string that is required to be searched or sorted. The functions involved are `LEFT$`, `RIGHT$` and `MID$`.

The format of each is as shown in Figure 12.3, if it is taken

that each uses the stated string variable and, using the stated parameter, creates a string, usually as another variable which is part of the original one. Whether it forms the start, middle or end of the original string is determined by the actual function used. **LEFT$** refers to the start of the original string, **MID$** the middle and **RIGHT$** the end. How many characters are involved in the new string is determined by the parameters of the function.

```
PRINT LEFT$(computer,3)          <ENTER>

> com


PRINT MID$(computer,4,3)         <ENTER>

> put


PRINT RIGHT$(computer,3)         <ENTER>

> ter
```

Figure 12.3 **Left$**, **Mid$**, and **Right$**

To demonstrate the fundamentals of these functions consider the following program lines, type them into your computer as a program and **RUN** the program.

```
10 INPUT "A word of three syllables"; word$
20 INPUT "Which letter position does the 2nd syllable start";start
30 INPUT "How many letters does it have";length
40 middle$=MID$(word$,start,length)
50 PRINT "The syllable is ";middle$
```

An inside-the-string Search

The function **INSTR** is of a different nature; its format is as shown in Figure 12.8 and can be used in the way shown in Figure 12.9.

```
10 CLS
20 DATA Steve,11 Freemans Close
30 DATA Andrew,3 Camden Road
40 DATA Carol,1 Dodd Street
50 DATA Elizabeth,82 Ashgrove Avenue
80 d=4
85 PRINT "Position of the word 'Close'";PRINT
90 FOR c=1 TO d
100 READ name$(c),street$(c)
110 PRINT INSTR(5,street$(c),"Close");
120 PRINT TAB(10) street$(c)
150 NEXT
160 PRINT
```

Figure 12.4 Use of the function **INSTR**

```
10 CLS
20 DATA Steve,11 Freemans Close
30 DATA Andrew,3 Camden Road
40 DATA Carol,1 Dodd Street
50 DATA Elizabeth,82 Ashgrove Avenue
60 INPUT "Start position of search";start
70 INPUT "Number, Road name or type";searchfor$
80 d=4
90 FOR c=1 TO d
100 READ name$(c),street$(c)
110 NEXT
120 FOR c=1 TO d
130 search=INSTR(start,street$(c),searchfor$)
140 IF search>0 THEN flag=1:GOSUB 180
150 NEXT
160 IF flag=1 THEN GOSUB 230
170 END
180 REM search routine
190 a=a+1
200 search$(a)=name$(c)
210 search1$(a)=street$(c)
220 RETURN
230 REM print criteria
240 CLS
250 PRINT searchfor$: PRINT
260 FOR b=1 TO a
```

```
270 PRINT search$(b);CHR$(32);search1$(b)
280 NEXT
290 RETURN
```

Figure 12.5 Development of the use of the **INSTR** function

Now experiment.

Chapter Thirteen

READ and DATA statements

The BASIC **READ** and **DATA** commands work together in all instances. The command **DATA** is a method for storing information which is used/manipulated from within the program. To all intents and purposes it is information that will remain the same every time the particular program is used. The command **READ** will fetch the information from the **DATA** statement and place the information stored in the **DATA** statements as contents in the appropriate variables, as in Figures 13.1 to 13.3.

```
50 DATA 1,12,13,14,14,19,18,17,16,15
60 READ number
65 MODE number: PEN number+2
70 FOR counter=1 TO 9
80 READ position
85 PRINT position;
90 PRINT TAB(position)"Screen Display"
100 NEXT
```

Figure 13.1 **DATA** and **READ** statements (i)

```
45 DIM word$(20)
50 DATA Delta,oscar,gama,scala," ",&
55 DATA " ",Ceta,alpha,toggle,swim
60 WHILE word$(count)<>"swim"
65 count=count+1
70 READ word$(count)
90 WEND
100 FOR c=0 TO count
110 PRINT LEFT$(word$(c),1);
120 NEXT
```

Figure 13.2 **DATA** and **READ** statements (ii)

```
50 READ number
60 CLS
70 PRINT "A/C no.","Surname":PRINT
80 FOR c=1 TO number
90 READ number,name$
100 PRINT number,name$
110 NEXT
120 PRINT
130 DATA 4
140 DATA 1423,Raven
150 DATA 6743,Sutton
160 DATA 2341,Smith
170 DATA 7453,Mills
```

Figure 13.3 **DATA** and **READ** statements (iii)

By sketching the above short programs, it is possible to understand the nature and uses of the **READ** and **DATA** commands. Examine the diagrams in Figures 13.4 to 13.6 and note the absence of a mention of actual **DATA** statements. There are two reasons. Firstly in a program **DATA** statements can be placed anywhere in the program that is convenient, this just happens to be usually at the beginning or the end. Secondly, if a structured diagram is written carefully the need for **DATA** statements will be fully documented at the point the information is **READ**.

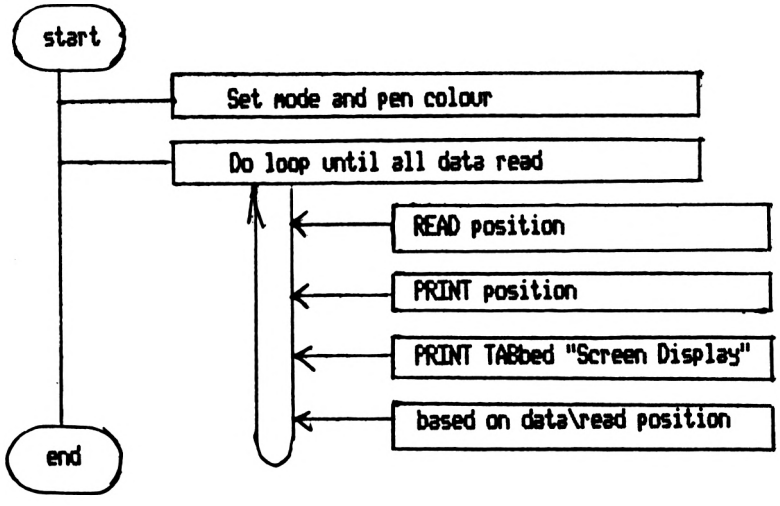


Figure 13.4 Diagram of program in Figure 13.1

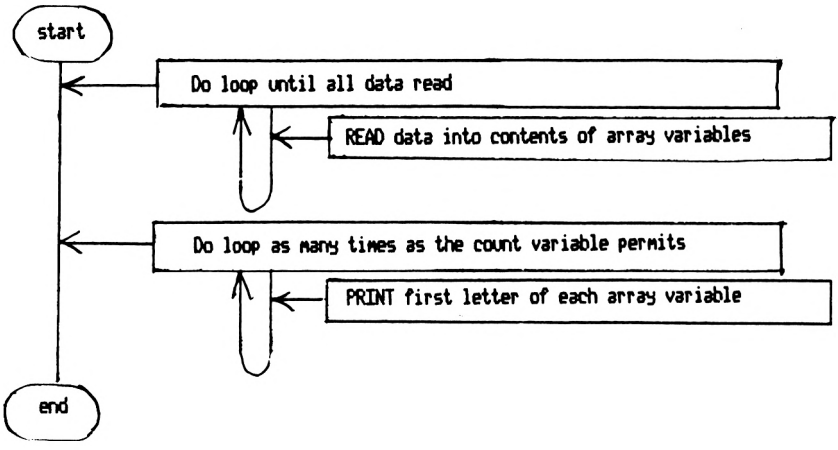


Figure 13.5 Diagram of program in Figure 13.2

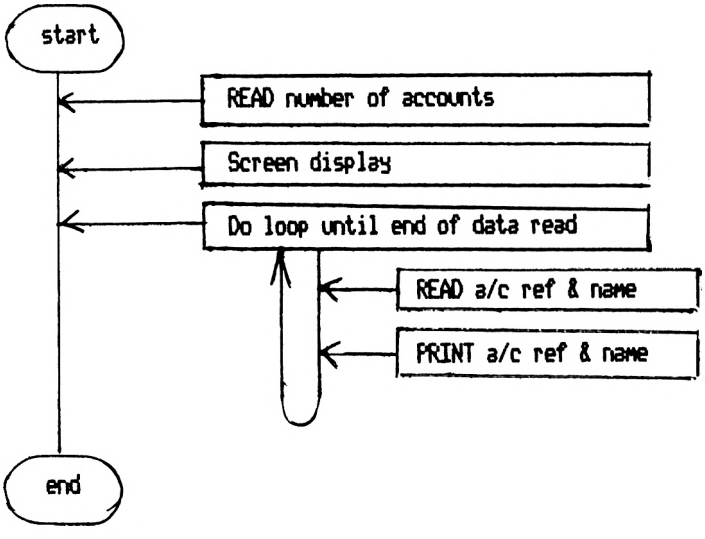


Figure 13.6 Diagram of program in Figure 13.3

The use of these commands is straightforward and follows these rules.

1) Each piece of information is separated by a comma(,).
2) Numerical and string information can be mixed in the **DATA** statement.
3) When **READ**ing **DATA** the numerical and string variables must match the particular type.
4) The first **READ** variable will be the first **DATA** piece of

information encountered in the program. The second **READ** variable will be the second **DATA** piece of information, and so on.

5) If you require to reset the **READ**ing of **DATA** to a specific **DATA** statement in the program the command word **RESTORE**, followed by the appropriate line number, is available.

One of the most practical uses of the **READ—DATA** facility is to repeat a particular subroutine on numerous occasions within a program, but with different parameters needed each time to cause a different effect. Thus there might be five different variables within this subroutine, but on each of the fifteen times the subroutine is executed each variable has as its contents a different number or string. It is a relatively simple programming technique to include the fifteen combinations of variable contents into fifteen **DATA** statements. At the start of executing the subroutine a **READ** statement fetches the new data and places it as contents into the variables concerned.

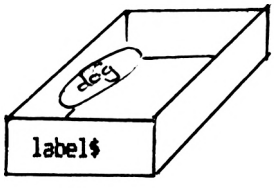
READ—DATA statements are a tool that can allow the imagination to take flight. Start by experimenting with some simple graphics, using variables and **DATA** statements to supply the parameters to create the shapes or colours.

Have a go!

Chapter Fourteen

Extending variables—dimensional arrays

The simplest form of variable is as shown in Figure 14.1: the variable is given a name/label and contents is placed in that created variable.




```
label$="dog"
PRINT label$
> dog
```

Figure 14.1 A standard string variable

In order to produce the telephone directory and several other of the short program examples it has been essential to use a different form of variable. This type of variable is characterised by a subscript or number within a bracket attached to the end of the variable name/label, for example, `Name$(2)`. Each of the individually coded variables with the same label can hold different contents. At the beginning of each program the number of each of these variables that are going to be used has to be stated or more technically dimensioned with the command `DIM`variable name/label(number of variables). These variables are referred to as dimensional arrays.

As demonstrated in Figure 14.2, arrays can be seen very much as a drawer in a filing cabinet. The label on the front of the drawer states the number of records to be held in the drawer (computer's memory). The drawer is then filled with suspension files each with a numbered label. Each file contains the content of that particular dimensioned variable. The

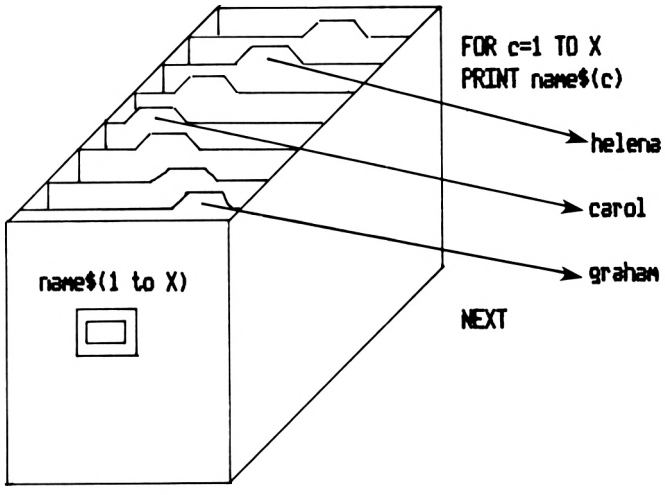


Figure 14.2 A filing cabinet analogy

beauty of this analogy is that, just like a personal file, each file can contain numerous array variables, for example, name, date of birth, address, etc.

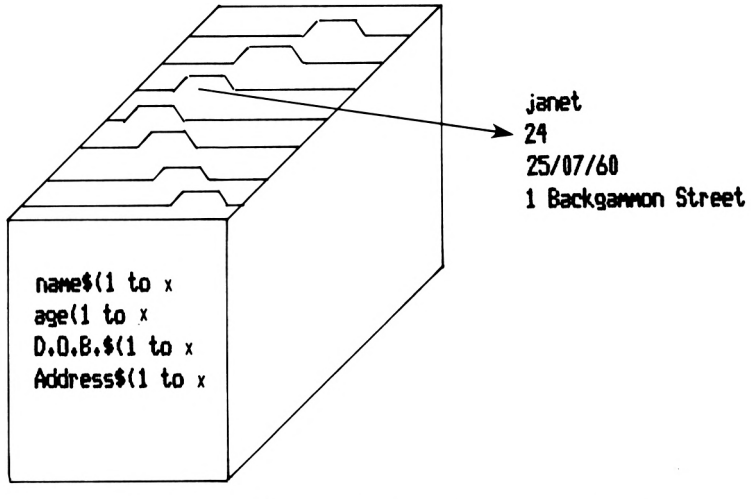


Figure 14.3 The file contains numerous array variables

So far we have considered only singly dimensioned array variables. Consider Figure 14.4 where instead of each piece of information concerned with each personal file being held as contents in different variables, a single variable can have a second dimension. Thus Figure 14.4 demonstrates 100 person-

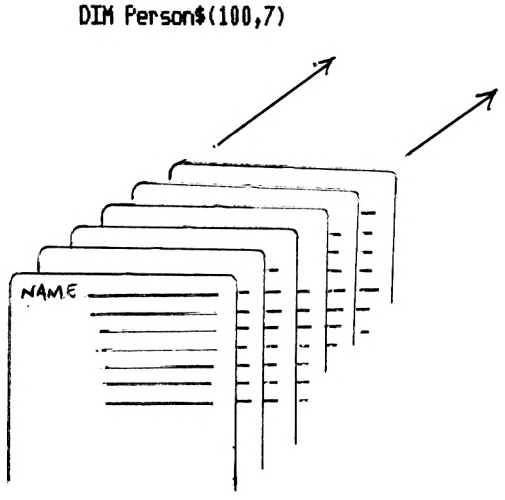


Figure 14.4 Further complications to storage of information

al files each with 7 pieces of information. This can now be taken a step further, as in Figure 14.5, where 100 personal files can hold 7 different areas of information, each area having 3 fields of information. This is a beautiful way of creating a program that can recall information accurately as it is organised very effectively. But do not get carried away: the problem with this particular programming technique is its extravagant use of the computer's memory.

Array variables can of course be both numerical and string in nature. Do not confuse them. Do not place numbers in a string array variable and expect to perform arithmetic with the contents. You can perform string manipulations but not arithmetic, and vice versa.

The structure of the program lines is:

```
10 DIM cell(3,3)
20 PRINT
25 FOR r=1 TO 3
30 FOR c=1 TO 3
40 PRINT cell(c,r);
50 NEXT
60 NEXT
```

The use of array variables within a loop make it exceptionally easy to load from and to a cassette or disk data file, print the contents of the variables on the screen, and compare the

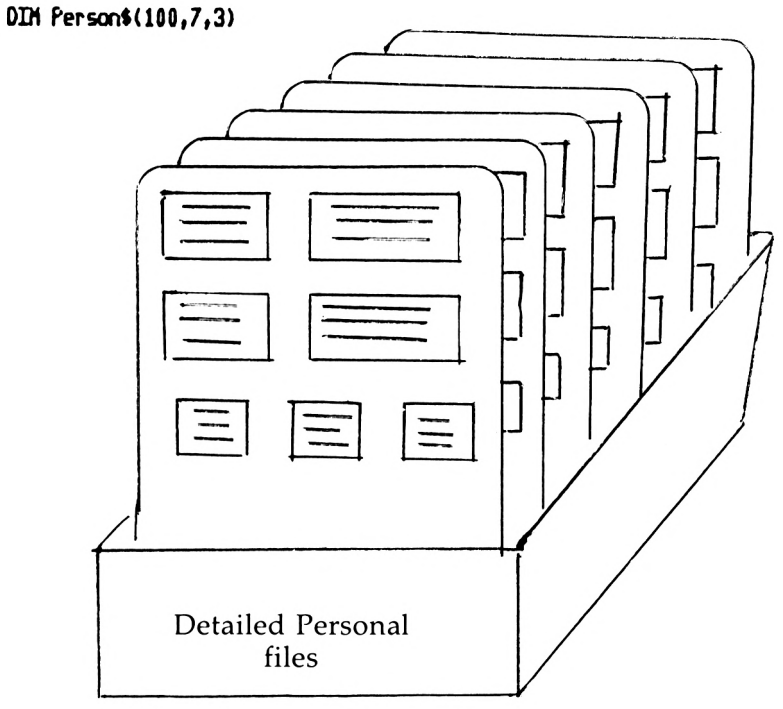


Figure 14.5 Further information storage

contents of each of the arrayed variables against a standard. The uses of array variables are very extensive and probably only limited by one's imagination in their application.

Chapter Fifteen

Viewing the file, formfil, screen format, and data entry

The cruz of producing a database program that can handle large amounts of information for the user is the method in which the information/data is put into the system to create the file. The actual method of entering the data will be determined by the programmed instructions, that is, whether the titles to the file of information are standardised or determined by the program user at the stage of setting up that particular file. This procedure is known as defining the file, i.e. the file definition. By a program employing this technique a simple database program can have numerous different applications.

A file creation routine to define each file structure needs to include the following questions to the user:

1) Number of fields of information? (maximum 6)
2) The title of field name one?
3) The title of field name two?

...and so on until the number of fields defined in question 1 have been given a title.

Each response of the user will be stored as the contents of a variable and stored on cassette or disk for use as a database's initialisation process for that particular application.

Professional software used in the world of commerce have these type of facilities as standard. During the database set-up procedure the user can define numerous options, and define the nature of every aspect of the file from the number of fields to which fields are displayed on the screen at any one time.

All of these professional file creation techniques can be achieved by BASIC programming as long as they are thought-

fully planned in a structured manner. The procedure would typically be:

1) Decide exactly the facilities available for use with the database.
2) Decide exactly the facilities that the user will be able to define.
3) Create a structure that will enable the user's responses to be stored as a data file.
4) Draw up the structure of the database program by means of structured diagrams.
5) Write the program in terms of 'first principles'; thus where the constants of the program are defined by the user the appropriate variable name is substituted. When the program is 'used' these will have been loaded from data file when the program is initialised.

Technical terms often used

Formfil is a technique sometimes used in professional software to set up the database's screen display. Essentially the program user controls the position of the cursor to define the position of information displayed on the screen. Each field of data can be individually placed on the screen, the cursor being positioned by the up, down, left, and right cursor keys. In terms of BASIC programming, the technique to produce this type of program would involve the testing of the position of the cursor on each axis of the screen, placing this information as contents into a variable. Each field of the database would have a set of coordinates to define its position on the screen, which would then be used when viewing the data in conjunction with the command `LOCATE`.

A screen mask is a conceptual term, describing the fields of information as set out on the screen. Imagine a blank form placed on the screen; the information contained, for example, on each personal record would then be displayed through this mask. By careful planning of the database program several different screen masks can be set up in order to view different aspects of each particular record.

A record is a collection of related fields, an example being the information contained about an individual person in a personal file.

A file is a collection of records of a similar nature.

A field is a single piece of information within a record, an example being the date of birth or street name.

Chapter Sixteen

Processing numerical variables–automatic record updating

Database applications such as a telephone directory are primarily concerned with the manipulation of string/character fields of information. In order to present a complete programming picture let us now consider the potential of using numerical fields within a database program. All numerical fields can of course have arithmetic performed on their contents. Imagine a screen of information contained within a database concerned with an individual's monthly earning over a one year period, as in Figure 16.1. The program user need only enter his or her tax allowance and make twelve entries of the amounts paid. The program will then calculate all the other values concerned, place them as contents in the appropriate variable, and subsequently display the results on the screen.

The program as shown in Figure 16.2 is merely the framework of a major database. Clearly the program would require a file handling facility to save the data entered by the user on a disk or cassette tape. In its present form the twelve monthly payments have to be entered one after each other *without* turning the computer off.

Examine the program listing carefully. Try the program out by keying it into your computer, **RUN** it, follow the instructions and watch it calculate the contents of the other fields.

For completeness follow the program while considering the structured diagram shown in Figure 16.3.

The use of the arithmetic functions is straightforward. The numerical variable names make up a formula. The contents of each variable is substituted into the 'formula' and the arithmetic is performed according to the rules of precedence:

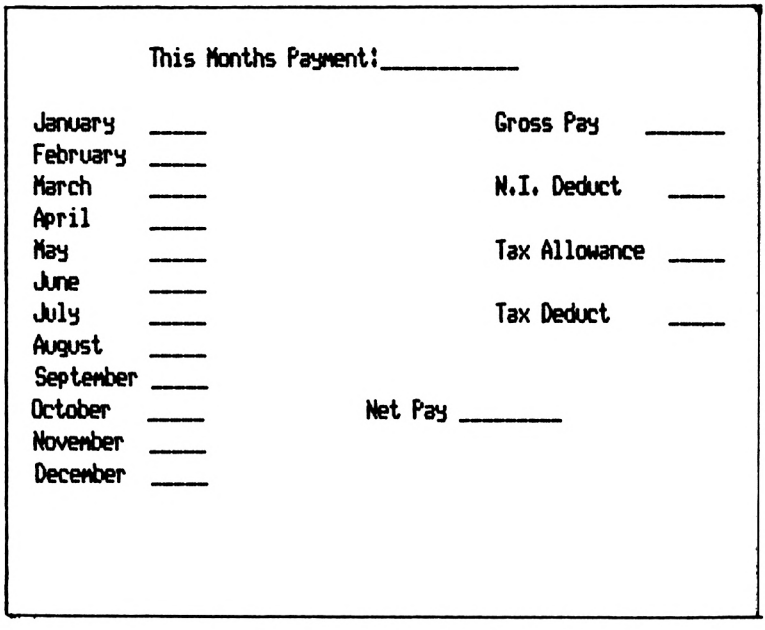


Figure 16.1 Screen display of a Personal Earnings database

```
<-INPUT "Tax Allowance"
<-c=1
  WHILE c<13
    <-INPUT payment(c)
     <-grosspay=grosspay + payment(c)
    <-NI=grosspay*0.06
    <-Tempcal=Gross-NI
    <-Taxable=Tempcal-Allowance
    <-Taxdeduct=Taxable*0.333
     <-NetPay=Temp-Taxdeduct
    <-c=c+1
  WEND
```

Figure 16.2 The calculating part of a Personal Earnings database

1) The arithmetic is performed from the left to the right.
2) Every element placed within brackets has precedence in the arithmetic, i.e. it is performed first.
3) All the multiplication elements are calculated.
4) All the division elements are calculated.
5) All the addition elements are calculated.
6) All the subtraction elements are calculated.

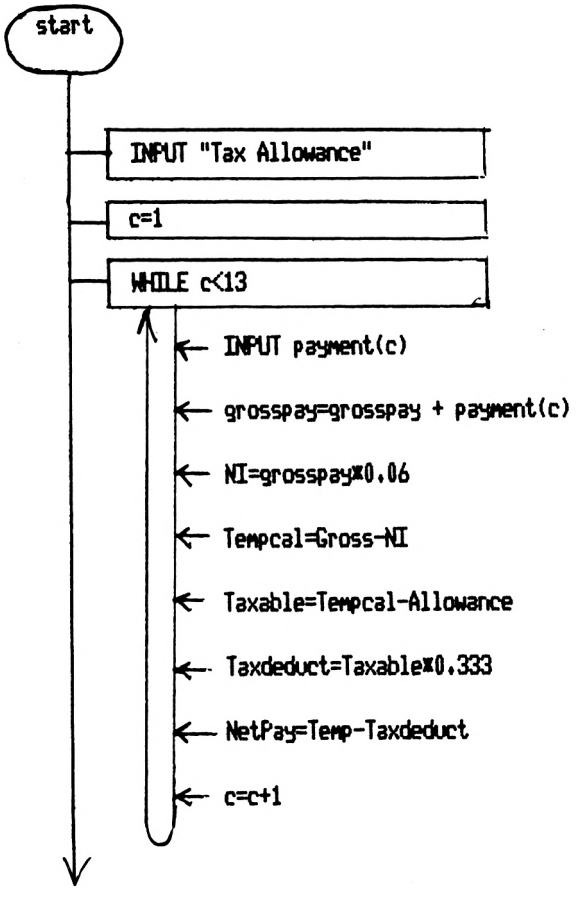


Figure 16.3 Diagram of program in Figure 16.2

The 'formula' takes the form of a LET statement. The variable which will have as its contents the result from the formula calculation is placed on the left of the equals sign:

```
Pay=Hourly rate*number of hours

Population of school=no. of staff+no. of pupils
```

In addition to performing the four basic operations of addition, subtraction, multiplication and division there are several further facilities or functions available. These functions include numerous methods of rounding off numbers, converting negative numbers to positive ones, and calculating various

trigonometrical functions either as degrees or radians. By the use of the **PRINT USING** command a range of different formats can be arranged. All of these facilities can be easily incorporated in a numerical processing routine of a program. Consult the computer's *User Instruction Manual* for details.

Programming functions are also useful. By incorporating a formula C=C+1 within a **FOR—NEXT** loop a simple counting machine has been achieved. This can then be taken a step further by writing the formula as C=C+A, where 'A' is a value **INPUT** by the user each time the **FOR—NEXT** loop is performed. The initial contents of the variable C is zero. The result is a totalling machine which will add together each of the values **INPUT** by the user. Figure 16.4 demonstrates this arrangement amply.

```
FOR b=1 TO 10

A=A+b

NEXT

Or

C=20

FOR b=1 TO 10

Total=Total+C

C=C+1

NEXT                same as  20+21+22+23+24+25+26+27+28+29+30
```

Figure 16.4 Sample of an adding sequence

The possibilities are endless, governed only by the imagination of the application. Once again the execution of an idea, its transformation into a program and subsequent application, is dependent on copious planning. It is most important to make sure that the processing routine is made to do the job it is meant to do. Create the processing routine as a separate element, the **INPUT** data as a separate routine, and the presentation of the processed results as a separate routine. To

make the task of processing easier use variable names that are very relevant to the function in the routine.

The example processing routine, calculating yearly pay after tax and national insurance deductions, is as set out in Figure 16.1 to 16.3, a running total system. This could be developed into a system where all the data is put in to the memory initially, the processing is then performed and the results finally displayed on the screen which is a logical and structured approach.

Chapter Seventeen

Ideas for applications

Applications for computers are as I see it the only way for the further development of the home computer. Today many people can manipulate their home computers to display patterns of numerous colours, both moving and static, and to animate characters around the screen, yet there are very few who actually apply their skills to programming needs around the home and small office. I guess the reason for this is not the lack of possible applications, but uncertainty about where to start.

To begin with, look at the ideas hinted at and examined in depth in this book, plus the areas of maintaining a cash book and other general accounting programs. Let us stay with the calculating angle. The question currently being asked in professional business software is 'what if?'. A condition or possibility is set and the possible resulting effect is calculated

```
If I earn £_____X_____ do I have enough money to
buy a new sports car?
Salary = £_____
Cost of Item = £_____
Deductions
Week 1  £_____      Week 2  £_____
Week 3  £_____      Week 4  £_____
Monthly Deductions  £_____
Yearly Expenditure W = Week 1 + 2 + 3 + 4 * 12 = ?
                   M = Monthly * 12 = ?
                   Exp. Total = W + M
                   ANS = X − Exp. Total
If ANS > Ø          BUY ITEM
```

Figure 17.1 Proposals of a 'What if?' dilemma

from the known facts previously entered into the program by the user. Consider the proposals put forward in Figure 17.1

The potential of array variables can be exploited in either of two areas: spreadsheeting or multi-dimensioned databases.

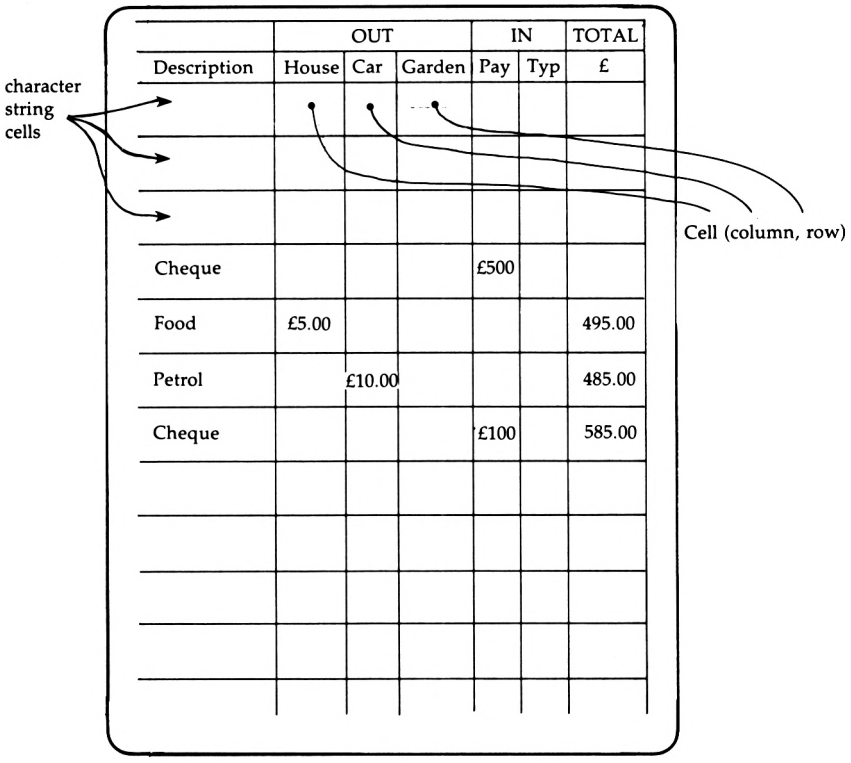


Figure 17.2 Sketch of a spreadsheet screen display—proposal for an application

A spreadsheet

By the use of two dimensional arrays, one numerical and one composed of characters, each with two dimensions, a matrix of cells can be created, each cell linked to an appropriate area on the screen. Figure 17.2 illustrates the theory. The blocks/squares on the screen can each be designed by the user to represent either the character variable or the numerical variable assigned to it. The column and row numbers can easily be displayed on the screen by a short program that would be similar to that shown in Figure 17.3.

```
* The use of PRINT USING command will be essential *

      DIM cell$(3,10),cell(3,10)
       FOR row increase by one
          FOR column increase by one
          PRINT cell( column,row)
          NEXT
      NEXT
       Repeat  process  for cell$(column,row) for those cells  that  have
      been  previously signalled as haveing character contents

      Enter values for calculation
      Calculation performed
      Answers and totals displayed
```

Figure 17.3 Possible programming methods

Each cell could be designated as being either a character cell or a numerical cell.

Each numerical cell could be assigned as an `INPUT` cell for the user to place values into the calculating process, or alternatively each numerical cell could have a `LET` formula assigned to it. The formula would consist of arithmetical manipulations of the cells designated as `INPUT` cells.

Clearly each area of the screen can only be assigned to either character cells or numerical cells.

Think about the idea and then try to develop it into a program application you could use.

A Multi-dimensioned database

By using an array variable `Person$(10,10)`, we create a card with 100 squares on it. The information held on it could be a different person on each row. Each column could be assigned to represent a different facet of information. Figure 17.4 demonstrates this approach.

NAME	AGE	STREET	TOWN	COUNTY	SALARY
GRAHAM	29	1 High St	Warwick	Warks	£9,000

Figure 17.4 A card of data—dimensioned array variables

A further development could be to make the array a three-dimensional one, `Person$(100,2,24)`, which could be read as 100 people having two pages each with 24 pieces of information on them; or `Person$(100,1,12,15)` which would mean 100 people having one page, which has 15 pieces of information printed on it and the last 12 pay cheques pinned to it. Pictorially this would be demonstrated as in Figure 17.6.

The combinations are endless and applicable to hundreds of different applications. The bones of each program would follow these lines:

1) Set up the structure of the database.
2) Define the function of each field.
3) Put data into the database, save onto disk or tape for long term storage.
4) Use the database to retrieve the information required.

File handling

By the use of cassette or disk files an exciting program could be

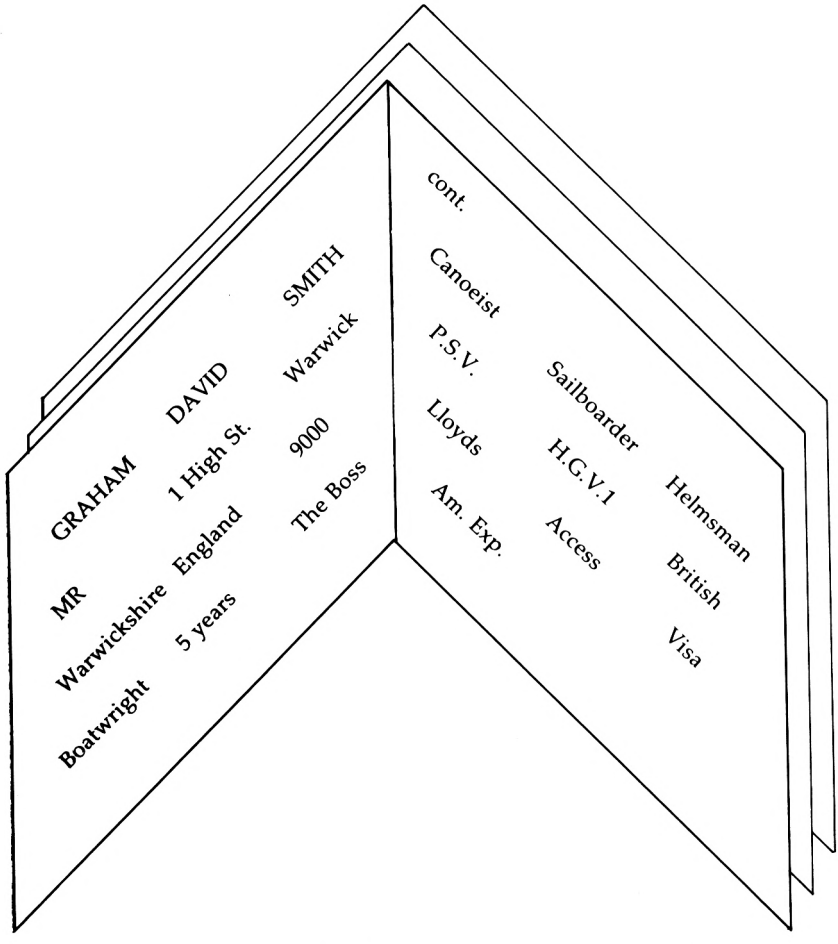


Figure 17.5 Two pages of information with 24 fields on each

developed. Unfortunately a cassette system is very much slower and will require manual manipulation to ensure the correct data is read from the tape at the correct time in the execution of the program. The theory goes that, for example, a personal record is set up with 20 pieces of information. That whole record is then stored onto either cassette or tape by means of a data file. The procedure is then repeated for the next person, and the next, and so on. Each time the information is stored on either tape or disk. The beauty of it is that the number of records that can be stored is limited only be the number of tapes or disks you can afford to buy.

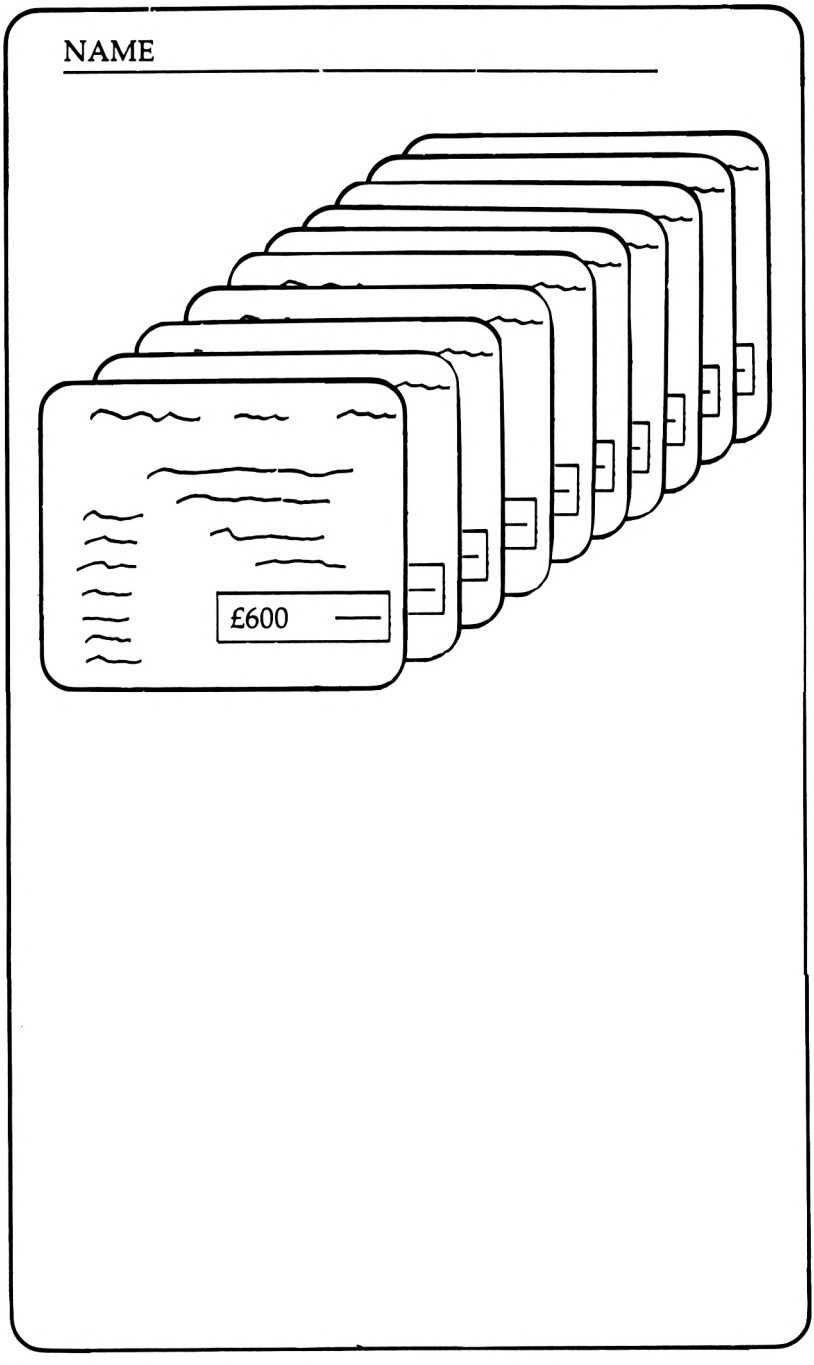


Figure 17.6 Pictorial demonstration of information held in dimensional array variables

Here is a small selection of possible directions. Take them, think about them, apply them to your particular needs. Begin with a diagram and sketches until you are satisfied that the program you have designed will fulfil the objectives you have laid down.

Chapter Eighteen

General principles and some reminders

When writing programs in BASIC you will normally find there are several ways of obtaining the same result. There is no right or wrong way in the majority of cases, so use the way you personally find the neatest and the most comprehensible.

Get into the habit of using **REM** statements frequently. Document your programs; you will only find out what I mean by writing some programs, leaving them for two months and then trying to read through them and understand the procedures you have written into each of the programs.

Programming guidelines

Use **INKEY$** commands embedded in **WHILE—WEND** loops rather than **INPUT** statements, minimising the times the program user has to press the <**ENTER**> key.

Use a main control element to direct the program from **GOSUB** routine to **GOSUB** routine. Use minor **GOSUB** routines for practical purposes such as 'Press any key' routines.

Link the **GOSUB** call and the start of every subroutine with a **REM** statement, to explain the function.

ON—GOSUB commands are very useful for driving a menu.

GOTO commands should be guardedly used. The direction of the flow of control is important. Forward jumping is a permissible action and will aid programming. Backward jumping can lead to never-ending loops and should be avoided as the control of the program's direction is taken away from the program user.

Readability of a program is important. **REM** statements will

help and so will the use of meaningful variable names. It is only too easy to assign a single character name as a variable; I am guilty of this myself. The important variables should be identified by words relevant to their functions, and in lower-case letters.

Validate your programs so that whenever the program user has a choice of one or more keyed responses, pressing any of the other keys will have no effect on the program. One simple way of achieving this is to use: `WHILE K$<>"B" OR K$<>"A" repeat the loop`. Thus unless A or B have been pressed the program lines within the `WHILE—WEND` loop will repeat *ad infinitum*.

There will be from time to time system errors that will cause BASIC to terminate execution of the program. However there are ways to validate against the program 'crashing' completely. This is done by the family of BASIC vocabulary that is concerned with error trapping.

`ERROR` and `ON ERROR` commands will cause the BASIC to perform some defined action on the result of a specific type of error or on all types of errors. Each error type is signalled by a specific code number. There are two variables `ERR` and `ERL` which return the error code number and the program line number where the error occurred, respectively.

When using `DIM`ensioned array variables only assign as many variables and dimensions as you are going to use. Arrays take up a lot of memory space.

When keying in the program instructions use a method of indenting the program lines which are inside a `WHILE—WEND` or `FOR—NEXT` loop; this will allow easier reading as one will be able to see where the repetitive parts of the program structure are.

Finally, become familiar with the tools you have at hand. As long as you take your time planning your programming projects, you can never be too ambitious; simply remember that you are using a home computer so memory space will eventually run out. But that won't be for a long time.

Appendix

Appendix:

The Telephone Directory

Part one: The option menu

```
10 REM Personalised Telephone Directory
20 REM Steve Raven   March 1985
30 REM Part one   Option Menu
40 CLS:PEN 3
50 FOR display=1 TO 22
60 PRINT TAB (display) "Telephone Directory"
70 NEXT
80 WINDOW 2,40,2,11
90 CLS:PRINT:PEN 2
100 PRINT"Do you require to:"
110 PRINT:PRINT"<A> Create or add to your directory "
120 PRINT:PRINT"<B> Use your directory"
130 PRINT
140 PRINT "Press the appropriate <key>"
150 WINDOW 2,40,13,21
160 CLS:PRINT:PRINT
170 PRINT "Remember you must have created a "
180 PRINT
190 PRINT "directory before you can use it."
200 PRINT
210 WHILE K$<>"A" AND K$<>"B"
220 K$=INKEY$
230 K$=UPPER$(K$)
240 PEN 1
250 WEND
260 IF K$="A" THEN RUN"Create"
270 IF K$="B" THEN RUN"Use"
280 END
```

Part two: Creating or adding to the directory

```
10 REM Create a telephone directory
20 REM or add new names and numbers
30 REM to a previously created
40 REM directory
50 REM Part two     Create
60 :
70 t=100:counter=0:record=0
80 DIM name$(t),surname$(t),phone$(t)
90 f$="Firstname":s$="Surname":ph$="Phone No."
100 MODE 1: PEN 1
110 :
```

```
120 GOSUB 210        :REM Option Menu
130 IF k$="A" THEN GOSUB 310
140                  :REM add to directory
150 IF k$="C" THEN GOSUB 620
160                  :REM create directory
170 GOSUB 960        :REM save directory
180 GOSUB 1190       :REM use or finish
190 END
200 :
210 REM Option menu on screen
220 LOCATE 1,2:PRINT"Choose the facility you require:"
230 LOCATE 5,4:PRINT"<C>reate a NEW directory"
240 LOCATE 5,6:PRINT"<A>dd to an OLD directory"
250 LOCATE 1,8:PRINT"Press either key <C> or <A>"
260 WHILE k$<>"A" AND k$<>"C"
270 k$=INKEY$
280 k$=UPPER$(k$)
290 WEND
300 RETURN
310 REM Add new phone nos to directory
320 CLS
330 LOCATE 5,24:PEN 5
340 PRINT"Insert data cassette in datacorder"
350 LOCATE 1,1
360 FOR delay=1 TO 1500:NEXT
370 CLS
380 OPENIN"data"
390 WHILE EOF=0
400 INPUT#9,name$(counter)
410 INPUT#9,surname$(counter)
420 INPUT#9,phone$(counter)
430 PRINT counter,
440 counter=counter+1
450 WEND
460 CLOSEIN
470 record=counter
480 WINDOW 1,40,1,23:CLS
490 PRINT f$,s$,ph$
500 counter=0
510 WHILE counter<record
520 PRINT name$(counter),surname$(counter),phone$(counter)
530 counter=counter+1
540 WEND
550 WINDOW 1,40,24,24
560 PRINT"Press key <c> to continue"
570 WHILE k$<>"C"
580 k$=INKEY$
590 k$=UPPER$(k$)
600 WEND
610 RETURN
620 REM create directory
630 WINDOW 1,40,1,25: CLS
640 PRINT SPC(10):PRINT"Information Entry":PRINT
650 PRINT f$,s$,ph$
660 WHILE counter<100
670 record=counter+1
680 WINDOW 1,40,11,11
690 IF k$=CHR$(32) THEN PEN 2
700 IF k$<>CHR$(32) THEN PEN 5
710 k$=""
720 PRINT"Record Number: ";record
730 WINDOW 1,40,5,9
740 PRINT SPACE$(80)
```

```
750 LOCATE 1,2
760 LINE INPUT; name$(counter)
770 LOCATE 14,2
780 LINE INPUT; surname$(counter)
790 LOCATE 27,2
800 LINE INPUT; phone$(counter)
810 WINDOW 1,40,17,25
820 PRINT"Press the <ENTER> key if data is correct"
830 PRINT:PRINT"If not press the <SPACE> bar"
840 PRINT:PRINT"If END of DIRECTORY press the <E> key"
850 WHILE k$<>CHR$(13) AND k$<>CHR$(69) AND k$<>CHR$(32)
860 k$=INKEY$:k$=UPPER$(k$)
870 WEND
880 CLS
890 IF k$=CHR$(13) GOTO 930
900 IF k$=CHR$(69) GOTO 920
910 IF k$=CHR$(32) GOTO 680
920 counter=99
930 counter=counter+1
940 WEND
950 RETURN
960 REM save the created directory
970 REM on data cassette
980 PEN 5
990 WINDOW 1,40,1,25: CLS
1000 WINDOW 1,40,11,11
1010 PRINT SPC(5);"Total number of records: "record
1020 WINDOW 11,30,20,24
1030 PEN 7: CLS
1040 PRINT SPC(1)"Save Directory"
1050 PRINT:PRINT"on data cassette"
1060 WINDOW 1,40,1,8
1070 PEN 1
1080 LOCATE 1,1
1090 OPENOUT "data"
1100 counter=0
1110 WHILE counter<record
1120 PRINT#9,name$(counter)
1130 PRINT#9,surname$(counter)
1140 PRINT#9,phone$(counter)
1150 counter=counter+1
1160 WEND
1170 CLOSEOUT
1180 RETURN
1190 REM Use directory or finish
1200 WINDOW 1,40,1,25: CLS
1210 PRINT:PRINT"Press the appropriate <key>"
1220 LOCATE 5,5: PRINT"<Q>uit"
1230 LOCATE 5,10: PRINT"<U>se the directory"
1240 WHILE k$<>"Q" AND k$<>"U"
1250 k$=INKEY$
1260 k$=UPPER$(k$)
1270 WEND
1280 IF k$="Q" THEN CLS
1290 IF k$="U" THEN CLS: RUN"Use"
1300 RETURN
```

Part three: Using the directory

```
10 REM Use Telephone Directory
20 REM Part three   Use
30 :
40 t=100
50 DIM name$(t),surname$(t),phone$(t)
60 counter=0
70 f$="Firstname":s$="Surname":ph$="Phone No."
80 MODE 1:PEN 1
90 :
100 GOSUB 200      :REM load data file
110 WHILE k<>4
120 GOSUB 380     :REM select facility
130 ON k GOSUB 550,710,950,1230
140 WEND
150 REM 1500-browse through directory
160 REM 2000-select and search
170 REM 3000-amend a record
180 REM 4000-quit
190 END
200 REM load data file
210 CLS
220 LOCATE 10,10:PRINT"Use Your Directory"
230 LOCATE 5,24
240 PRINT"Insert data cassette in datacorder"
250 LOCATE 1,1
260 FOR delay=1 TO 2500:NEXT
270 LOCATE 1,1: PEN 2
280 OPENIN"data"
290 WHILE EOF=0
300 INPUT#9,name$(counter),surname$(counter),phone$(counter)
310 PRINT counter,
320 counter=counter+1
330 WEND
340 CLOSEIN
350 record=counter
360 PEN 1
370 RETURN
380 REM select facilties
390 WINDOW 1,40,1,25: CLS
400 PRINT TAB(8)"Facilities Available"
410 LOCATE 5,3
420 PRINT"Choose each option by pressing the"
430 PRINT"appropriate number and hit the <ENTER>  key"
440 LOCATE 8,7
450 PRINT"<1> Browse through directory"
460 LOCATE 8,9
470 PRINT"<2> Select and search"
480 LOCATE 8,11
490 PRINT"<3> Amend a record
500 LOCATE 8,13
510 PRINT"<4> Quit
520 LOCATE 8,15
530 INPUT k
540 RETURN
550 REM browse through directory
560 CLS
570 PRINT f$,s$,ph$
580 WINDOW 1,40,3,18
```

```
590 c=0
600 WHILE c<record
610 PRINT name$(c),surname$(c),phone$(c)
620 page=c MOD 14
630 IF page=0 AND c>0 THEN GOSUB 1280
640 IF page=0 AND c>0 THEN WINDOW 1,40,3,18: CLS
650 c=c+1
660 WEND
670 WINDOW 8,32,20,23
680 PRINT"End of Directory"
690 GOSUB 1280
700 RETURN
710 REM select and search
730 CLS
740 PRINT"Do you want to select by: "
750 PRINT:PRINT"<A> First name
760 PRINT:PRINT"<B> Surname"
770 WHILE k$<>"A" AND k$<>"B"
780 k$=INKEY$
790 k$=UPPER$(k$)
800 WEND
810 PRINT
815 name$="zz":surname$="zz":f=0
820 IF k$="A" THEN INPUT"Which first name";name$
830 IF k$="B" THEN INPUT"Which surname";surname$
840 PRINT:PRINT "   ";f$,s$,ph$
850 WINDOW 1,40,10,21
860 c=0
870 WHILE c<record
880 IF k$="B" THEN 900
890 IF name$=name$(c) THEN f=1: PRINT c;name$,surname$(c),
    phone$(c)
900 IF surname$=surname$(c) THEN f=1: PRINT c;name$(c),
    surname$,phc
910 c=c+1
920 WEND
925 IF f<>1 THEN LOCATE 10,8:PRINT " No Record Found "
930 GOSUB 1280
940 RETURN
950 REM amend a record
960 CLS
970 PRINT"Please select record for amending by: "
980 GOSUB 750
985 IF f<>1 THEN RETURN
990 WINDOW 1,40,22,24
1000 INPUT"Input the number of the record you want to change";n
1010 CLS: PRINT"Re type the whole record please."
1020 WINDOW 1,40,10,21: CLS
1030 PRINT n;name$(n),surname$(n),phone$(n)
1040 LOCATE 2,4: INPUT;name$(n)
1050 LOCATE 12,4: INPUT;surname$(n)
1060 LOCATE 25,4: INPUT;phone$(n)
1070 WINDOW 1,40,1,25
1080 CLS
1090 PEN 7
1100 PRINT"Save Directory on Data Cassette"
1110 LOCATE 1,22
1120 OPENOUT"data"
1130 c=0
1140 WHILE c<record
1150 PRINT#9,name$(c)
1160 PRINT#9,surname$(c)
1170 PRINT#9,phone$(c)
```

```
1180 c=c+1
1190 WEND
1200 CLOSEOUT
1210 PEN 1: PAPER 4
1220 RETURN
1230 REM quit
1240 CLS
1250 LOCATE 10,10
1260 PRINT"That's All Folks!"
1270 RETURN
1280 REM press any key routine
1290 WINDOW 1,40,25,25
1300 PAPER 1: PEN 3
1310 CLS
1320 PRINT TAB(8)"Press any key to continue"
1330 k$=INKEY$: IF k$="" THEN 1330
1340 PAPER 4: PEN 1: CLS
1350 WINDOW 1,40,1,25
1360 RETURN
```

Index

addition 128, 129
addition sign 49
advantages of CPC 664/464 3–4
alphabetical order 109–10
'amend a record' subroutine 95–7
applications for computers 132–6
arithmetical calculations 7–8
arithmetical operators 8
arithmetic functions 127
array variables 121–3, 133, 140
arrow cursor key 43
ASCII Codes 36–8, 104–5, 109
AUTO command 15
automatic insert mode 43
automatic line numbering mode 15

BASIC language 3–4, 6, 8, 31, 34, 86
blank spaces 17, 36–7, 41, 48
browse facility 92

calculation 6
caps lock 106
cassette recorder 3, 6
cassettes 26
cassette storage systems 4, 66, 137–8
cassette tape 38
CAT command 17–18, 22, 27
character cell 134
CHR$ (32) code 37
CHR$ function 104, 107
CLOSE IN sequence 91
CLOSEOUT command 38
colons 9–10
colour 33, 55, 87
columns 36
commands 32
commas 9, 36
conditional frameworks 71
"Create" program 33, 40, 52
CTRL key 10–11, 15
cursor 6, 9, 32, 35–6, 125

data cassette 14
DATA command 116–117
DATA file 60, 138
data file management 14
data recorder 14
DATA statement 116–19
DATA tape 59
debugging a program 41–2
  order of operations 41
default stream 36
diagrams 73–9
dimensional arrays 52, 110, 120
DIM variable command 120
direct command mode 5–6, 10, 42, 47
  applications 11
disk storage 4, 14
division 128, 129
DRAWR command 12, 13
draw statements 12

editing facilities 43
EDIT 'line number' command 42
electronic calculator 6
END command 77
ENTER key 5–7, 9–10, 15–16, 104
equipment 3
'error messages' 4, 17, 42
errors 41–2, 140
error trapping 140
ESC key 10–11, 15, 17, 22, 40

file creation 124
file definition 124, 126
file handling 95, 127, 137–8
flag variable 95
floppy disk 38
flow chart 73
flow of control 77

formfil 125
FOR-NEXT loop 32, 34, 38, 51, 55, 72, 130, 140
FOR statement 32, 41
FOR-NEXT statement 33

GOSUB command 34–5, 59, 89, 139
GOSUB-RETURN command 34
GOTO command 38, 139

IF-THEN command 34, 38, 58, 93
INKEY$ command 33, 52, 60, 86, 139
INPUT cells 134
INPUT command 38, 53–5, 57–60, 139
inside-the-string search 113
INSTR function 113
instructions 3–5, 14
  number of characters 10
iteration 33

jumping 139

keyboard 4
K-string 33

LEFT$ function 112–13
LEN function 105
LET statement 11, 48, 129, 134
LIST command 16–17, 22, 40, 42
LOCATE command 35–6, 125
LOWERS facility 106, 109
lower case 6, 33, 49, 106, 109–10, 140

memory 122
MID$ function 112–13
monitor 3
multi-dimensioned databases 133, 135
multiple choice 76
multiplication 128, 129

NEXT command 32, 141
'No record found' instruction 93
numerical cells 134
numerical fields 127

ON ERROR command 140
ON-GOSUB command 59–60, 139
ON k GOSUB command 89

OPENOUT command 38
option menu 15–18, 31, 33, 40, 51, 59, 66, 99
output stream 35

PHONEHOME file 27
PLAY key 17
'Press any key' routine 93, 95, 98, 139
PRINT#9 command 38
PRINT command 7, 9, 11, 35–6, 58
printer 6
PRINT SPACES command 36
PRINT SPC command 36
PRINT USING command 130
processing routine 130
program loading 66
programs 3, 5
  interactive with user 57
punctuation 9

'Quit' option 89
quit subroutine 97
quotation marks 49, 55

readability of program 139
READ command 116–117
READ statement 117, 119
READ variable 118–19
READY sign 59
record 52
  definition 126
REM statement 19, 24, 31, 34–5, 139
repetition 34, 70, 76
repetition of tasks 71
repetitive loop 89, 91, 93
RESTORE command 119
RETURN command 34–5
return key 42, 43
RIGHT$ function 112–13
RUN command 10, 16, 22, 27
RUN "PHONEHOME" facility 40

SAVE "Create" command 22
SAVE "PHONEHOME" command 17–18
SAVE "Use" command 26
screen displays 66–72
  planning 66
screen mask 125
screen MODE 36
scrolling 40

searching 108
'select and search'
subroutine 93–5
'select facility' subroutine 89, 91
selection 34, 70
selection of tasks 71
semi-colons 9
sequence 70
sequences of operations 74–6, 85
sequences of tasks 70
sequence type instructions 34
SHIFT key 7, 9–11, 15, 43
skills 4
software 124–5
sorting 108–10
space bar 37
SPACE$ command 36, 107
SPC command 36
'speed write' facility 66
spreadsheets 133–4
$ symbol 27, 49, 103–7
start up time 66
string variables 49, 106, 108
  contents 103
strings 103
  length 105
STRING$ facility 106–7
structured planning 65–72
subroutines 24, 35, 78, 86–7, 89–92, 95
subtraction 128, 129
symbol key 9, 31
symbols 8
syntax 4, 6
syntax errors 7, 41

TAB command 7, 36, 107
telephone directory 14–15, 31, 34, 47, 65, 89–91
  adding to 18–22
  creating 18–22, 28, 52, 66, 80
  part three 88
  using 23–7, 53, 66
title page 66–7
top-down approach 73, 80
trigonometrical functions 130
two dimensional arrays 134
typing errors 5, 17

upper case 33, 49, 106, 109, 110
UPPER$ command 86, 106, 109
User Instruction Manual 5, 6, 130

validating 17, 60
variable 'display' 32
variables 42, 47–8, 50, 54
  contents 47

WEND statement 33
WHILE command 33, 41
WHILE-WEND loop 34, 53, 59–60, 93, 139, 140
WINDOW command 36
windows 36
'word processing' 107

STRUCTURED PROGRAMMING ON THE AMSTRAD COMPUTERS 464, 664 and 6128

In order to give instructions to a computer, they must be presented in a very precise form so that the micro can understand and hence execute them.

The Amstrad CPC 6128, 664 and 464 computers have several advantages for creating well-structured programs: they communicate in a form of BASIC that is easily read, their documentations are comprehensive, and both the 664 and 6128 have the distinct advantage in having disk storage — a time-saving element in loading and storing programs.

The book takes the reader through every element of creating a program. Planning by the use of Top Down diagrams is one of the many features of the book. Sections are provided on: Introducing the CPC 6128, 664 and 464, Familiarity Breeds Confidence, The Principles of BASIC, From Little Blocks to Structured Programs, and Handling the Text — The Key to Information Storage.

The Author

Stephen Raven is a freelance Data Control Consultant. Prior to that he was a lecturer in Computer Studies at Redbridge Technical College in Essex.

ISBN 0-7447-0034-5

9 780744 700343

£9.95

STRUCTURED PROGRAMMING ON THE AMSTRAD COMPUTERS CPC 464, 664 and 6128 Stephen Raven

AMSTRAD CPC

MÉMOIRE ÉCRITE
MEMORY ENGRAVED
MEMORIA ESCRITA

https://acpc.me/