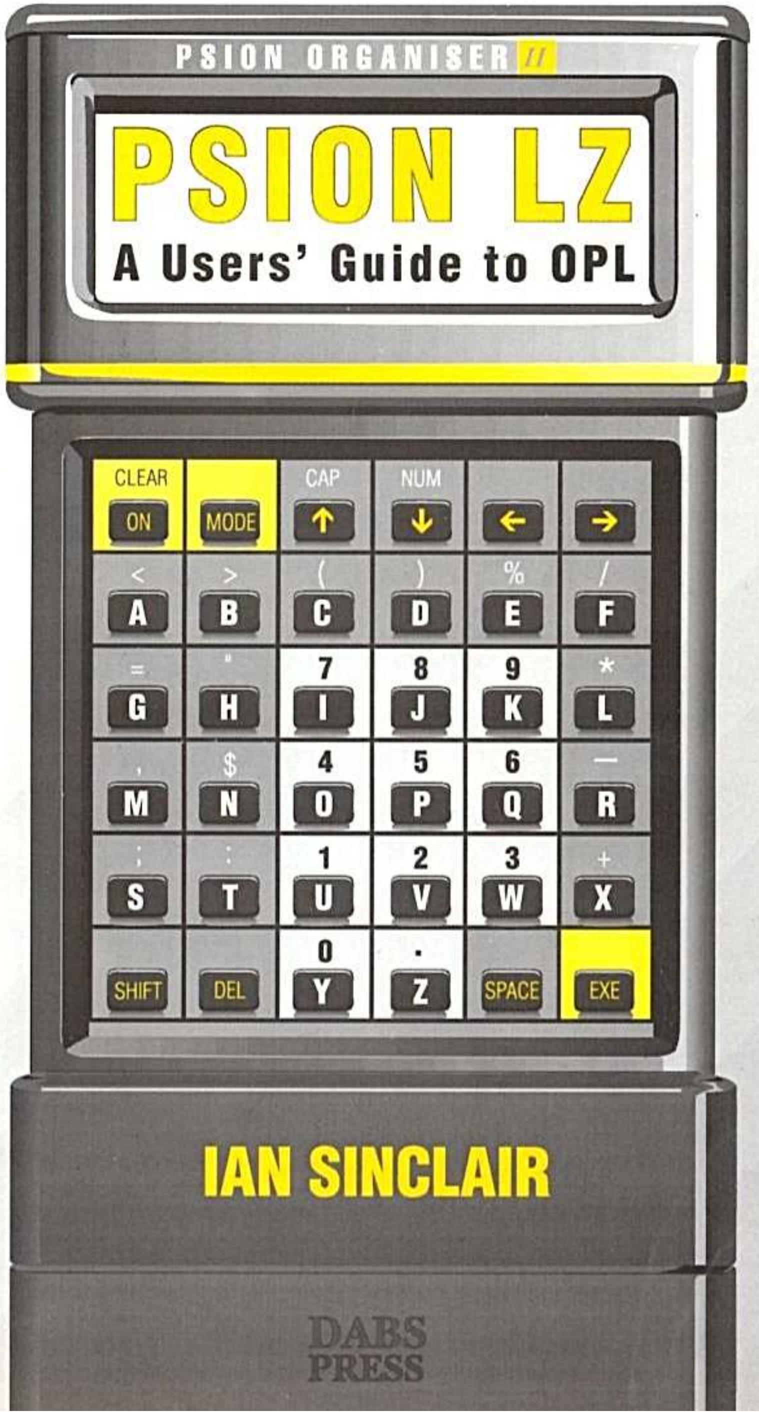
A Dabhand Guide



PSION LZ
A Users' Guide to OPL

IAN SINCLAIR

DABS PRESS

Psion LZ

A Dabhand Guide

Ian Sinclair

DABS PRESS

Psion LZ: A Dabhand Guide

© Ian Sinclair 1990
ISBN 1-870336-92-5
First Edition, first printing May 1990

Editor: Sharron Fellows
Typesetting: Bruce Smith, Andrew Wygladala
Cover Artwork: Atherton Clare Designs

This book was typeset in 10/12pt Palatino using an Apple Macintosh desktop publishing system.

Published by Dabs Press, 22 Warwick Street, Manchester, M25 7HN. Telephone 061-773 8632, fax 061-773 8290, to whom all enquiries should be addressed.

Printed and bound in the UK by BPCC Wheaton, Exeter, Devon EX2 8RP.

Contents

LZ

Chapter Summaries

LZ

Chapter 1

Explains some of the new features of the Psion LZ Organiser, including the international time-zones and telephone dialing codes, as well as an explanation of the stopwatch and 'daylight-saving' functions. You are introduced to the organiser programming language (OPL) - a variety of BASIC, and how it works in principle. You are shown exactly how to type in a program, save it and run it. There are the first of several example programs for you to try throughout the chapter.

Chapter 2

Introduces the idea of a variable and assigning some value, number or phrase, to a name. Number types, integer and float, are introduced, and we see how number variables can be used. The other type of variable, the string, is described and the important differences between a number held as a string and a number held as a number-code are emphasised. The next topic is the INPUT action of entering data into a program while it is running, and this is followed by the single-character type of Y/N input that can be so useful. The remainder of the Chapter is concerned with numerical work with operators, expressions and formulae, since so many users of the Organiser require these actions. The important point of precision of number is fully explained here.

Chapter 3

One of the very important computing actions, the loop, concerns repetition of instructions, and this is introduced here. Repetition goes hand in hand with testing, so that instructions are repeated only for as

long as is required. This Chapter also deals with the important string functions, such as counting characters, slicing letters and words, and the conversions between number form and string form. Conversions between upper and lower case of letters are also dealt with, and here too is information about how to sort words into their alphabetical order one of the most-wanted programming actions. The Chapter ends by introducing the array, the way in which a list of data can be treated as one entity while still allowing individual items to be picked out.

Chapter 4

This Chapter contains some of the most important sections about programming. Many beginners to programming believe that learning the programming language is the most important part of programming, but this Chapter points out that planning is the key to programming in any language.

Menus are an important help to good program structure, so that the menu ability of the Programmer is fully explored. The design of a simple program from the foundation level is fully explained using an example which requires no specialised knowledge. By showing the development of this programs from outline idea to finished code you can see how you need to proceed for yourself in designing your own programs. This Chapter also explains the important principle of procedures.

Chapter 5

Files and filing of data forms the subject of this Chapter, since filing is one of the strong points of the Organiser. The keywords of file, Record and Field are explained, and the differences between filing to RAM and filing to Datapack are emphasised.

Files are then introduced by using simple examples that require the minimum of keyboard use, and the actions that can be carried out on files are exemplified by a totalling program. The design of filing programs is dealt with, and attention is given to how a record can be changed, and how records can be selected by name, by dates or by other criteria. The extremely good statistical actions of the Organiser are shown used for analysis of data in files.

Chapter 6

Covers briefly a number of topics that have not been dealt with earlier. The date functions are treated in more detail, showing how dates can be used in integer number form, and the conversion to and from this form. The error-trapping system is then examined, showing how you can avoid the indignity of having a program stop because of an entry error, and warning of the limitations of the system. The Organiser has better facilities in this respect than most desktop machines. The beep of the Organiser can be controlled, and this also is dealt with. The remainder of the Chapter is concerned with program instructions which are more specialised and therefore less frequently used. After this, its all yours!

Conventions

A number of simple conventions are used throughout this Dabhand Guide to make things easier to follow, we hope!

All listings and Psion responses are displayed in a different typeface to that used for the text of the book. The typeface is Courier, and an example is shown here:

```
pro7x2:
print "Hello"
```

Preface

LZ

The Psion Organiser is by now a well-proven product which has enjoyed considerable success as a truly pocket-sized machine with very considerable capabilities. The earlier models, however, were always relatively handicapped by their screen size which permitted only two lines of text. The new LZ models with a four-line screen, allow for a very considerable expansion of what is possible with a portable computer.

The features that made the earlier models so successful have been retained, and new facilities added. These are international dialling codes and times in 400 cities in 150 countries around the world, a notepad for short pieces of text, and a stop-watch action for the built-in clock. More importantly, however, the four-line screen also allows the LZ Organisers to be programmed more effectively, making them an extremely formidable portable machine that can challenge anything else around once you have come to terms with the keyboard.

This book does not deal with the built-in actions that are common with the older machines, though the newly introduced facilities are covered in some detail. The main emphasis is on the programming of the LZ or LZ64 model in its built-in OPL programming language so as to enhance the use of the machine now that the four-line screen makes such efforts much more worthwhile. The examples have been chosen to reflect the wide variety of applications for the Organiser which, because of its versatility, is used as much as a desktop machine as it is as a portable. All of the programming examples have been typed and used on the Organiser itself, then copied to a PC machine for printing on a Star LC24-10 dot-matrix machine. For this reason, all examples are short and can be tried and used with the minimum of effort. The PC Link which allows Organiser text to be transferred to and from a PC machine has been dealt with in an Appendix. This is because the manual for the PC-Link is by no means easy for the casual user as distinct from the experienced computer owner.

I am greatly indebted to David Atherton and Bruce Smith of Dabs Press for providing the Psion Organiser LZ64 on which this work was done, and to Psion for further technical information, particularly on the PC Link itself.

Ian Sinclair
Autumn 1989

1 : Setting Up

LZ

I shall assume that by the time you read this, you have already loaded batteries into the LZ Organiser and checked out the method of switching on and off. Remember that if you have a PC Link connected, the Organiser will be switched on whenever the PC is switched on. Therefore, if you want to switch the Organiser off when the PC is on, you need to disconnect the link. I shall also assume that you have made use of the facilities that are available on the older machines, such as the calculator, calendar, database, diary, alarm and the choice of languages. For the new user of the Organiser these facilities are clearly explained in the manual, and there seems little point in repeating the instructions here.

However, for the benefit of users of the older machines who are upgrading to the LZ, and for new owners who may only have seen one of the older models demonstrated, or used by a friend, this book starts with a description of the facilities that have been added to the machine in order to enhance its value and make good use of the larger screen space. The most immediately noticeable of these facilities is the time display in the top right-hand corner of the screen, but the main additions are those which appear in the menu.

International Data

When you switch the LZ Organiser on by pressing the ON button, you will see the screen choice shown as:

```
Find          Save          Diary
Calc          Time          Notes
World         Alarm         Month
```

Pressing the key that is marked with the yellow downwards-facing arrow a few times will shift the pattern of words so that additional lines appear:

```
Prog          Xfiles        Utils
Comms         Off
```

A flashing block, the 'cursor', appears on the first letter of the first word, Find. Selecting from this 'menu' can be done in either of two ways:

1) Use the keys that are marked with the arrows to move the cursor to the item you want, then press the EXE key (bottom right) or

2) Press the key marked with the first letter of the word that you want. For example, to obtain international data, you should press the 'W' key or press the EXE key while the cursor is over the word World. This will produce a display such as:

```
Monrovia
Liberia
Tue 13 Jun 8:26a
Dial: 010 231
```

which shows a city, country, relative time and the international telephone dialling code for making calls to that city.

The time is shown as relative to Greenwich Mean Time (GMT), with no adjustments for any summer-time changes. This means that when you select cities in the UK, such as London or Nottingham, you will see a time which in summer will be one hour earlier than the time on a local clock. Obviously, since not all countries alter clocks for summer-time, and those that do so do not necessarily alter the clocks at the same time as we do, this cannot be programmed into the Organiser. One item that you do have to program in, however, is the UK alteration to summer-time.

When you select the Time option, you should see a display of the form:

```
Tue 13 Jun 1989 Wk24
9:43:36 am (D)
London
United Kingdom
```

in which the time is followed by a (D) to indicate that daylight saving is being used. When the clocks are set back in October, you should not manually alter the time. Instead, press the MODE key and then the right-arrow key three times to reveal the daylight-saving choice from the set:

```
Stopwatch     Timer  Set     Daylight-saving
```

Having made this choice, select Off from the On/Off choice that is then revealed. Similarly, if you have started using your Organiser in the Winter, on the day in March when the clocks are set forward you should select Daylight-saving and then On to advance the time by one hour.

In this way, the time as displayed on the clock when you select Time is always in agreement with your clocks, but the GMT is maintained unaltered so as to ensure that the time displays for other cities around the world will be correct. Never adjust the time manually by one hour for summer-time start or finish because this will put all the international times out by one hour during the summer. If you want to telephone an office abroad, you will have to know whether or not they have set their clocks on, but since you know the GMT for that region you can easily add an hour if this is needed.

The international telephone dialling codes are usable from any country outside the one that you are dialling, assuming that you are dialling from a country that uses modern telephones with international dialling. If you are dialling back to Great Britain from another country, the code you need is 044, which is not in the Organiser list, but which you can add in a note.

The Notepad

The LZ Organiser provides for Notepad use selected from the Notes item on the main menu. Selecting Notes provides you with a screen which shows a quill pen and the current time on the top line, and the word Notepad: at the start of the next line. You can now enter text by pressing the buttons – remembering that the layout is in alphabetical order rather than in the pattern that is used by typewriters and desktop computer keyboards.

- The text will be printed in one line on the screen, scrolling across as you add more words, unless you press the EXE key to provide a new line. You can use the EXE key in the same way as you use the Carriage Return on a typewriter, or the ENTER/RETURN key on a computer.
- To get a capital letter, hold down the SHIFT key (bottom left) and press the CAP key which is marked with the yellow up-arrow, then release both keys. The alphabetic keys will now provide capital letters. Repeat the SHIFT-CAP action to return to lower case (small) letters.
- To move down a line, press the down-arrow key, to move up a line subsequently, press the up-arrow key.
- If you want to switch to numbers, then either hold down the SHIFT key as you press the keys that are marked with numbers above or hold down SHIFT and press NUM (then release both) to make the keys give numbers rather than letters when they are pressed without using the SHIFT key. When you are set for NUM, the cursor no longer flashes, and other letter keys such as G,H,M,N etc give the punctuation and arithmetic symbols that are marked above the buttons.
- The DEL key deletes the character (letter, digit or punctuation mark) to its immediate left. There is no provision for deleting the character at the cursor position.
- Use the ON/CLEAR button to leave the Notepad action.

When you have typed a note – remembering that a brief note that you can read quickly is much more useful than a long note that you might ignore – you can press the MODE button for a menu (press ON/CLEAR to return to the Notepad). This menu offers:

```
Find Save Load New Home End Calc Sort Number Password
Print Dir Copy Delete Zap
```

which looks bewildering when you first see it, particularly since you can only ever see three or four of the choices on the screen at any one time and the menu repeats as you use the arrow keys to scroll across the screen. The uses of these options, ignoring the order above, are as follows.

Home and **End** place the cursor at the start and the end, respectively, of your note so that you can get to these points without the need to keep pressing the arrow keys.

Find allows you to type a word, or part of a word, so that when you press the EXE button the word or part-word will be found in the text of the note and marked with the cursor. Pressing the EXE button again continues the search until you see a notice to the effect that there are NO MORE ENTRIES.

Save allows you to store the note as a file in the memory. This lets you delete the note from the Notepad but recall it later. If you do not use Save, but do not delete the note you have typed either, it will remain in the Notepad until you do delete it or until the memory is cleared (by taking too long about changing the battery, for example).

- When you opt for Save, you will be shown a filename of:

  ```
  A:Notepad
  ```

  which you can use by pressing EXE. However, it is more likely that you will want to use your own filename such as JUN13 or CHAS or EXHIB – something that reminds you of the substance of the note.

New allows you to pick a new filename for a new Notepad. You will be reminded if there is any note in the current Notepad that you have not saved.

Zap deletes all trace of any note in the Notepad you are using. Saved material is unaffected.

Dir produces a list of saved notepad files. You are shown the A: reminder first, in case you want to use B: or C: (if you have additional packs fitted). Pressing the EXE button then displays the files of that set. Press ON/CLEAR to return. Load allows you to see a list of current files and select one to put into the current Notepad and view.

Delete allows you to delete a file, as distinct from the current note (which can be deleted with Zap). Always look at a file before you delete it.

Number will produce numbered lines for you as you type your note. The numbering starts at the second line, under the filename, so if line numbering is important you should always start a note of this kind by pressing the EXE button. The numbering is recorded when you Save the file and will appear again when you Load it.

Sort will sort lines alphabetically, using the first word in the line. The sorting is carried out regardless of case (ie, capitals are not sorted separately) and this is an excellent way of putting a set of names into alphabetical order.

Calc is used when part of a note contains numbers and you want to find the result of a function such as the SUM of the numbers, how many number ITEMS exist, the MAX or MIN number size, or statistical results like MEAN (average), VARiance or STandard Deviation.

- To use Calc, each number has to be entered with an '=' sign before it, and the function must be entered with an '=' sign after it. Selecting Calc (you will need to press EXE to do so) will then produce the figure for the function result.

Password allows you to specify a password without which the Notepad cannot be read (except by someone who understands how the Organiser stores its data). If you take this option, you will be asked to type the password twice – the word does not appear on the screen except in the form of a row of asterisks to indicate the number of characters.

If you subsequently forget your password, you will never be able to recover the information in the data file.

- Don't keep your credit card PIN number on a Notepad, even with password protection, unless you are very confident that no-one could ever know your password. Also remember that anyone can delete a file with a password – see below.
- Don't use obvious passwords like your date of birth, car registration, spouse's name etc.
- If you subsequently want to change a password, select Password, type the existing password, then the new one (twice). Use the EXE button in place of a new password if you want to cancel the password.
- Password affects only the file you apply it to – other files can be loaded and viewed without any request for a password.
- If you delete a file with a password, the need for a password is also removed, and another file that you create with the same name will not require a password.

Print allows you to send a Notepad file to a suitable printer (a serial printer). If no printer is connected, you will get the ERROR DEVICE MISSING message as a reminder, and you will have to press the SPACE key to get back to normal working.

Copy allows the contents of a Notepad to be copied to another memory pack (usually from A: to B: or C:). Remember that data should not be copied to a Datapack unless you intend it to be permanently stored, because you cannot erase a Datapack as easily as you can erase memory.

The Stopwatch

The Stopwatch facility is an addition to the Time menu on the LZ models only. To use the Stopwatch:

1) Select Time and then press MODE
2) Select Stopwatch – you will then see the screen display:

```
STOPWATCH
00:00:00.00
```

3) To start timing, press EXE.

4) To stop timing, press EXE again.

5) You can restart and re-stop as many times as you like, getting a cumulative time, by pressing EXE repeatedly.

6) You can reset the Stopwatch to zero again with the DEL button after pressing EXE to stop timing.

- If you press DEL when the Stopwatch is counting, the display shows the time at which you pressed the DEL key, but the count continues (a flashing dot reminds you of this). To return to the current count, press DEL again.
- If during counting, you press the SPACE key, this prints the elapsed time as a lap time. Then when you next press the SPACE key, you will get the next lap time, ie, the time that has elapsed since the previous time the SPACE key was pressed.
- As usual, the ON/CLEAR button returns to the normal Time menu on the first press, and to the main menu on the second press.

For the new owner of an Organiser, this description of the added features will be a good introduction to the methods that the Organiser uses for its other actions. With the guidance of the Manual, you will be able to work with these other actions quickly and with few difficulties. The remainder of this book is now concerned with the more difficult topic of programming the Organiser so that it can be used for applications that are outside the range of the preset ones that are built in, or added by way of inserted packs.

Programming

Now the reasonable question to ask at this point is – why should you program the computer for yourself when for a few hundred pounds

you could have as many professionally-written programs as you might need? The figure of a few hundred pounds is one good reason in itself if you are a home user, though for business purposes this might be chicken-feed. The one single overwhelming answer to this question is that only by programming for yourself do you get exactly what you want.

Suppose, for example, you run a small mail-order catalogue group, and you want to keep tabs on who has ordered what, how much has been paid, and when delivery is made. Now there isn't a commercial program to do this, as far as I know. You could buy a set of business program packs such as Spreadsheets and Databases and Accounts packages, but you'll be overwhelmed by them. They all need a lot of learning to use, and they all do much, much, more than you need. A business accounts program, for example, keeps a dozen ledgers going, has entries for aged debtors and all sorts of other accountancy terms, and you spend most of your time trying to work out which bits are needed. Since you can't dispense with any of them, though, you have to try to use them all, just as if you were running a medium-scale business.

By spending just a fraction of that time and a lot less money on learning to program for yourself, you could have your own program, tailored to your own requirements, doing what you need of it. Remember that when you buy a program written by someone else, the program is in charge, and decides how you have to proceed. When you write your own program, you are in charge, and you decide what the program produces. If you don't need VAT, the program doesn't calculate it. If you need a list of customers in alphabetical order, or in order of how much they owe you, your program can be made to give that – it's all up to you to decide what you want and arrange for it to be supplied. Control, then, is the main reason for wanting to program, whether you use your Organiser for business or for pleasure.

There's another reason, which has nothing to do with business but a lot to do with curiosity. You can use a computer as you might use a car, putting up with its odd little ways, but never doing anything to understand them. Using a computer in this way is never entirely satisfying – you always feel that the machine will have the last word. Just as by understanding what makes a car tick (or run smoothly as the case may be) you can drive it better and avoid breakdowns; you can also, by learning more about the computer become able to make better use of it. Computers are still at an early stage of development. It would not be an exaggeration to say that small computers are today in much the same state as cars were when the Model T Ford was introduced. In any case, the more you know about the machine, the better you can drive it. In addition, programming is a very considerable aid to thinking. When you learn to program, you also learn to break a problem down into manageable pieces, and work on the pieces. If you are programming for some business reason, you'll learn a lot more about your business from writing the program, than you imagined possible. If you program for a hobby reason, then both your hobby and your computing will come on in leaps and bounds. Programming is the most stimulating mental activity that there is, and you don't have to start at Professor level to get a lot from it – just watch a class of 8-year olds at work with a computer!

Programming Languages

A program for a computer is a set of instructions, and a programming language is concerned with how these instructions are written. When you use a computer, as distinct from programming it, you use 'commands'. In a program, the words of command are written in the sequence in which we want them to be carried out, but they are not carried out until we issue another command. The difference is important. A direct command, like using Calc and typing:

```
        2.55*3.62
```

is carried out by typing the instruction and then pressing EXE. If you want to repeat the command, you have to go through all of these steps again.

A program, by contrast, can consist of a number of separate steps, written once, and which can be executed as many times as you like just by using one command. The words or codes that are used to mean instructions in a program are what make up the programming

language, along with the way that the words must be used, which is the 'syntax' of the language.

One very useful language, called BASIC, was originally devised to solve the problem of teaching the language FORTRAN. The letters of BASIC mean Beginners All-purpose Symbolic Instruction Code, and that's what it originally was – a simple language intended to serve as an introduction to programming, and modelled on one of the great original computer languages, FORTRAN. The advantages of the original BASIC was that it was simple to learn, but close enough to the methods of FORTRAN to make the conversion easy. BASIC could be made in interpreter form (see later), so that mistakes could be easily and quickly found and corrected. Finally, BASIC could be written in code that took up only a small amount of memory. It was for precisely these reasons that when microcomputers became available, they featured BASIC as their programming language.

Since then, BASIC has developed a long way, and it's no longer just a language for beginners. As the language grew, it acquired more features from other languages, and without losing its simplicity. Consequently, it soon became a language in its own right, not just a path to an almost-forgotten FORTRAN. Because of the intensive use of BASIC versions on small computers, the language became a general – purpose one, good for all kinds of programs whether your interests were in accounts, science, engineering, text editing, or whatever. Other languages tended to remain specialised, good for only one or two selected purposes, while BASIC grew to fit the needs of users. Nowadays, more people can program in BASIC than in any other language, and they don't necessarily learn it so that they can learn another language. After all, you don't learn English so that you can later learn Icelandic or Sanskrit. The Organiser Programming Language (OPL) is a modern variety of BASIC.

Compilers and Interpreters

Like many other languages, BASIC can be 'interpreted' or 'compiled', but is nearly always interpreted on small computers. Whatever language you use to express your program in, it has to be converted into number codes before it can have any effect on the computer. Interpreting and compiling are two methods of carrying out this conversion.

When a language is interpreted, each instruction is taken, converted into machine code, and then executed before carrying on to the next instruction. In practical terms, this means that each instruction word of the language calls up a set of number-codes to do the work.

A compiled language, by contrast, converts all of the instructions of a program into a large machine code program, which is very often recorded on disc rather than being run at the time. The action of translating from high level language into machine code is called compiling. Once the program has been compiled, it is a machine code program which will run when you start it. OPL is a compiled language, which means that you write it in ordinary text, and this text file is translated into a machine code file, which is the part that runs. This distinction between the plain text source code and the translated machine code is very important to remember when you start to work with OPL.

Principles of Programming

On the surface, programming is quite straightforward. You type a list of instructions in the order that you want the machine to carry them out. You record this set of instructions, which is the source code. Finally, you make the translator convert them into machine code, which is also stored so that you can load it in when you need it. That's all.

It would be just as simple as this if the machine could understand English, but it can't. OPL, like each other computing language, allows you to use a limited number of instruction words. This number can be large, 130 or so for OPL. It's still very small, however, compared with the thousands of words that a 'natural' language like English uses, and one of the problems of learning programming is trying to express what you want to do with such a limited number of words. This means that

you have to break down any problem into small pieces that can be tackled by using a few of these 'reserved words' of OPL. The other snag is that each reserved word or keyword has to be used in a very precise way, the syntax of the word. If you don't use the word correctly, the instruction cannot be carried out, and you will get an error message that reads 'syntax err' to draw your attention to it. Programming means precision, then, and how you use and place words is as important in a programming language as it is in Latin – which is why Latin scholars make good programmers. In short, programming teaches you to analyse problems, and to tackle them with precision – and that can't be bad training for anything.

We'll see as we progress how all of this can be done, but at the start we can't really illustrate problem solving with OPL when we don't know much OPL. The best way to get started is the practical way, because when you have done something for yourself, you remember it better than when you have only read about it. I'll assume, then, that you can carry out the instructions in this book as and when they appear. The first step is to practice how to start using OPL when you want it, and to write, store and translate programs that you have typed.

Printing Out

When you think of what the Organiser does when you run programs on it, you soon conclude that its actions fall into four categories. One is accepting data that is typed in, another is printing data on the screen or on paper. A third category is calculation, arrangement, or other work. The fourth is memory saving or loading. Very often the third type of action, the actual computing, takes the least time. The first two are of very considerable importance, because they are the actions that you are most directly concerned with. In this Chapter, then, we'll look now at how you can write program instructions that result in something being printed on to the screen, or on to paper. Until you have some mastery of this particular craft, you can't very well tell whether your computer is doing anything useful or not.

The first step, though, is to check that you can save and load an OPL program. That's not because there's any difficulty about it, but because it's different from the techniques that you use with other types of programs. The simplest test involves a very simple type of program, but it will give you the confidence that you need. It can be a very heart-breaking experience to spend a lot of time typing in a program, and then find that it vanishes when you switch off because it was not correctly recorded. If anyone tells you that they have never done it, don't believe them. We all have, and one good method of losing data in this way is to be unfamiliar with the recording system.

Start, then, by selecting Prog from the main menu. This gets you to a menu that consists of:

```
Edit     New    Run
Print    Dir    Copy
Delete
```

and to start with you need to select New so as to start a new file. You will be asked for a filename, subject to the usual rules of a maximum of eight characters starting with a letter or digit. Try 'test1' for your first effort, so that your screen will show:

```
New  A:test1
```

and you can press EXE to enter this. The screen will now show:

```
test1:
```

with the flashing black cursor, the marker that shows where anything that you type will be placed, immediately after the colon. Now you're ready to type a program. OPL allows you to type program instruction words in either lower case or upper case letters. The EXE key is used to take a new line when you are typing.

Now type the words '`rem try`'. You can use the DEL key to delete letters if you make a mistake, but you are not likely to make any mistakes in something as simple as this. It doesn't matter whether you type 'rem' or 'REM'. This is a command word for the computer, and no matter whether you type it in upper or lower case, the computer will

deal with it correctly. Check that this looks correct, and then press the EXE key. The effect of this is to place this instruction line into the memory of the machine. As you type each character you will see it on the screen, in lower case or upper case, at the cursor position. When you press the EXE key, the cursor moves to the next line down, ready for the next instruction and enters the previous line into the memory. Even if your 'line' happens to take up more than one line on the screen, don't press EXE until you have finished and want the complete line placed into the memory. Now type the rest of the lines, as shown below, remembering to press the EXE key after you have finished typing each line:

```
test1:rem try
rem another
rem again
```

The program is now complete, and what you see on screen is called a listing. Listing an OPL program means that the computer prints on the screen whatever you have stored in its memory – OPL uses the Edit command to produce such a listing, and you have to select which listing you want if you have created more than one program. Now to make the recording. First of all you have to press MODE, which brings up the menu:

```
Tran Save Quit Find Home End Zap Xtran
```

of which you select Save, meaning that the program file is to be stored in the memory until you delete it. The filename will automatically appear. Press EXE to see this recorded, so that the Organiser returns to the programming menu. You can now prove that the program has been recorded by selecting Dir from this menu. When the screen shows:

```
DIR OF OPL
Dir  A:
```

you can press EXE again to bring up the file name of test1. This is shown as the line:

```
Opl test1 41
```

in which the 41 is the number of memory spaces used to store the program. To check that the program can be returned into use return to the Prog menu by pressing ON/CLEAR, select Edit from this Prog menu and use the filename of test1 and reply Y when you are asked to confirm. This will remove the text from your screen. However, the next time you select Edit and use the filename of 'test1' you will see the program return. To try this, press MODE and select the Quit option, then select Edit as before.

Now if this were a 'serious' program, we could use the Tran option of the Programming menu to translate the program into something that would run, but there is no point with this example. The keyword REM simply means Reminder, it's a note to yourself and it does not make the computer do anything. Because of that, it isn't translated, so there is no point in using Tran in this example. The save and load actions are the ones that we want to make certain of at the moment.

Once you can reliably save your programs to the memory, check that they are in the memory, and then re-load them, it's good to know that a few seconds more work will save your efforts so that you don't have to type them all over again. You can save a partly-typed program, or a program which has mistakes and won't run, so that you can come back to it another time and finish it. A further step is to save your text files on a PC disc by using the PC Link to transfer the files into the desktop machine, and that point is noted in Appendix A.

Let's now take a look at the difference between the sort of direct command that you would use along with Calc, and a program instruction. We have already seen how Calc will multiply the numbers 2.55 and 3.62. If you want to make a program of this you would select New, then type a file name – we could use 'pro1x1' for this one. When you press EXE, the filename should now appear on the first line with a colon following it – you do not need to type the colon, and problems will be caused if you do. Press EXE to take a new line and now type the program which should appear as:

```
pro1x1:
print 2.55*3.62
get
```

The word 'print' can be in upper case (PRINT), or in lower case (print). You have to start with print (or PRINT), because a computer is a dumb machine, and it obeys only a few set instructions. Unless you use the word 'print', the computer has no way of telling that what you want is to see the answer on the screen. It doesn't recognise instructions like 'GIVE ME' or 'WHAT IS', only these few words (about 130 of them) that we call its 'reserved words', 'keywords' or 'instruction words'. PRINT is one of these words. So that you can recognise these reserved words more easily in this book, I shall print them in upper case (capital) letters in the text from now on. You know by now, however, that you can type them in either upper case or lower case, and the machine will take the same course of action. The program examples will show mainly lower case lettering, because that's the way they were typed.

The PRINT instruction, then, is the way that we get the computer to provide information to us. In OPL, however, PRINT has the effect of putting information on the screen for only a moment, and then returning to the menu. The last line in the program consists of the keyword GET which makes the computer wait for a key to be pressed. This has the effect of holding the screen as it is, allowing you to see the result of the program instruction. When you press a key, any key, you will see the menu return.

To see it all happen, return to the menu by pressing MODE and select Tran. The translation is over in a moment, and you will see the Save instruction appear so that you can ensure that both the source code and the translated version will be saved together. Press EXE to execute the Save and return to the menu, now select Run. The program name will already be in place, and when you press the EXE button the program, such as it is, will run and produce the result 9.231 on the screen. Pressing any key then returns you to the menu.

All computer uses involve inputs, processes, and outputs. The hard part about programming is not learning the language of OPL, it's learning how to get what you want from these three actions. For the moment, we'll leave inputs aside, and concentrate on outputs and a bit of processing. Processing means doing whatever we want to do with numbers or words. It can be as simple as adding two numbers, or as complicated as putting a set of names and addresses into alphabetical order of surname. Summing it up, it means all of the actions that make a computer so interesting and so useful. Let's start programming then, with the arithmetic actions of add, subtract, multiply and divide. Computers aren't used all that much for calculation, but it's useful to be able to carry out calculations now and again. In addition, you are much more likely to recognise the sort of instructions that use numbers than the ones which are used to work with words.

```
prolx2:
print 5.6+6.8
print 9.2-4.7
print 5.06*6.08
print 7.06/1.4
get
```

The listing above ('prolx2') shows a short program which will print some arithmetic results. The process here consists of four bits of arithmetic, each with an output. Take a close look at this, because there's a lot to get used to in these few lines. To start with, the lines are arranged in the order in which the instructions will be carried out. The next thing to notice is how the number zero on the screen is slashed across. This is to distinguish it from the letter O. The computer simply won't accept the 0 in place of O, nor the O in place of 0, and the slashing makes this difference more obvious to you, so that you are less likely to make mistakes. The zero that you see on the keyboard is not slashed, but it is on a different key, and is differently shaped. Type some zero's and O's on the screen so that you can see the difference.

Now to more important points. The star or asterisk symbol in the third line is the symbol that OPL uses as a multiply sign. Once again, we can't use the 'x' that you might normally use for denoting multiplication because 'x' is a letter. There's no divide sign on the keyboard either, so OPL, like all other computing languages, uses the

slash (/) sign in its place. This is the diagonal line which is on the same key as the letter 'F' (press the SHIFT key along with the 'F' key).

So far, so good. The program is entered by typing it onto the screen, just as you see it, using GET for the last line. You then:

1) Press MODE and select Tran.
2) Press EXE again when the Save option appears.
3) Select Run and press EXE to see the program work.

- If you do not Translate a program, but only save the text, it cannot be Run. You can, however, save a program at any stage to edit later, and then Translate it when it is complete. If you Edit a program which has already been translated, it must be translated again before you can Run the new version.

When you Run this particular example, the last line should give you some idea of how precisely OPL can carry out its arithmetic, showing 11 places of digits beyond the decimal point. Later, we'll look at how numbers like this can be rounded off to a smaller number of places. When you follow the instruction word PRINT with a piece of arithmetic like 5.06*6.08, then what is printed is the result of working out that piece of arithmetic. The program doesn't print 5.06*6.08, just the result of the action 5.06*6.08. When the program stops printing, you can end it by pressing the EXE key to return to the menu.

This program has done two of the main computing actions, process and output, for you. The only input has been of the numbers in the program itself, and we can't alter any of these numbers in the program without altering the program by editing. This requires you to be certain that the program is retained in the memory. If you record another program with the same filename as an earlier one, OPL assumes that you are replacing the old program with a newer version, and it deletes the older version. Now try writing a program of the arithmetical type for yourself, and see how OPL carries out the calculations and displays the answers.

All of this is useful, but it's not always handy to get just a set of answers on the screen, especially if you have forgotten what the questions were. OPL allows you a way of printing anything that you like on the screen, exactly as you type it, by the use of what is called a 'string'. The listing below ('pro1x3') illustrates this principle.

```
pro1x3:
print"2+2=";2+2
print"2.5*3.5=";2.5*3.5
print"9.4-2.2=";9.4-2.2
print"18.48/2.2=";18.48/2.2
get
```

In each line, some of the typing is enclosed between quotes (inverted commas) and some is not. You must not forget to enter the semicolon (;) which separates the two sections of each line, because if you do so you will get an error message, and the cursor will be placed at the first character following the missing semicolon. Can you see how very differently the computer has treated the instructions? Whatever was enclosed between quotes has been printed exactly as you typed it. Whatever was not between quotes is worked out, so that the first line, for example, gives the unsurprising result:

```
2+2=4
```

There's nothing automatic about this. If you edit the first line to read:

```
PRINT "2+2=";5*1.5
```

then you'll get the daft reply, when you Run this, of 2+2=7.5. The computer does as it's told by the program, and that's what you told it to do. Only a loony would believe that computers could take over the world! The important point about this example is that it shows how to make a program display what is being done. As before, the command word PRINT has to be used to make things appear on the screen, but by using quotes, we can make the computer print whatever we want, not just the results of some arithmetic. Try making the computer print some answers for yourself, using this form of program.

With all of this accumulated wisdom behind us, we can now start to look at some other printing actions. PRINT, as far as OPL is concerned, always means print on to the screen. For activating a paper printer (hard copy, it's called), there's a separate form of instruction, LPRINT, that is not so straightforward to use because it requires the Comms Link to be present and correctly set up, see Appendix A. The Comms Link is also needed if you want a program listing on paper. These instructions are not useful to you unless you have a printer connected. If you use them without a printer connected and switched on, the computer will report the absence of the connection and you will have to press the SPACE key in order to return to normal service.

Now try the program listed below. When you Run this program, the words appear on three separate lines. This is because the instruction PRINT doesn't just mean 'print on the screen'. It also means 'take a new line', and start at the left-hand side. You will also find, of course, that when you have more than four PRINT lines like this, then when the words on the screen reach the bottom line, all the lines move up, and the top line disappears, scrolling out of sight. You can make the scrolled lines reappear by using the arrow keys.

```
pro1x4:
print"OPL - "
print"the way "
print"to program"
get
```

In this example ('pro1x4'), the words have been placed between quote marks, and they have appeared on the screen just as we typed them, but with no quotemarks showing. This, then, is the sort of programming that is needed when you want to display instructions or other messages on the screen. The real problem, as you'll see when you try it, is of getting the messages to look really neat. Nothing looks worse than printing which has words split, with half of a word on one line and the rest on the next line. If you type a long line following PRINT" then you will see the screen shift sideways so that the new words do not disappear beyond the edge, and you can use the left or right arrow keys to see the whole of the printing when you come to run the program. It will not, however, appear like this when the program runs unless you use one of the other methods of printing to the screen such as VIEW or DISP (see later).

Even at this stage, it's possible to make your printing look neat with some care, by not going over the edge of the screen. Suppose, for example, that you have some long paragraph that you want to type in, using several long PRINT lines. Type the PRINT" part, then type the words that you want, and continue typing without touching the EXE key until the screen starts shifting sideways. If you are in the middle of a word, then erase this word by using the key, type the closing quotemark, and then press EXE. Start another line now, using PRINT", and then type the rest of the message in the same way. This way, you will never split a word across a line end. Try it!

Now the action of selecting a new line for each PRINT isn't always convenient, and we can change the action by using various punctuation marks and instruction words that we call 'print modifiers'. Edit the program that you have just tried, adding the semicolons in the first and second lines as shown below in 'pro1x5':

```
pro1x5:
print"OPL - ";
print"the way ";
print"to program"
get
```

The effect of a semicolon following the last quotemark in a line is to prevent the next piece of printing from starting on a new line at the left-hand side. When you run this program, all of the words appear in one line. It would have been a lot easier just to have one line of program that read:

```
PRINT"OPL - the way to program"
```

to do this, but there are times when you have to use the semicolon to force two different print items on to the same line. We'll look at that sort of thing later in program examples. Meantime, look also at how I have placed a space between the last letter and the last quote mark in

the first two lines. The semicolon doesn't just order the computer to prevent a new line being taken, it also forces it to place one item right up against another. If you left no spaces, the phrase would be printed as 'OPL – the wayto program'. Try removing the spaces, and see for yourself.

Rows and Columns

Neat printing is a matter of arranging your words and numbers into rows and columns, so we'll take a closer look at this particular art now. To start with, we know already that the instruction PRINT will cause a new line to be selected, so the action of the listing below ('pro1x6') should not come as too much of a surprise.

```
pro1x6:
cls
print"This is OPL"
print
print"working for you"
get
```

The first line contains a novelty, though, in the form of a new instruction that, strictly speaking, we don't need yet. The instruction CLS clears the screen, and makes the printing start at the top left-hand corner of the screen. This is done automatically in any case when you Run, but the CLS instruction will be useful later when we want to wipe clean a screen by an instruction in the program. Another point about the program is that the PRINT instruction, with nothing to be printed, will cause a blank line to be taken. There are other ways of doing this, as we'll see, but as a simple way of creating a space, it's very handy. Now try for yourself a program which will put words on different lines like this. Remember that you have only four lines to play with on the screen.

```
pro1x7:
at 10,1
print"x"
at 9,2
print"xxx"
at 8,3
```

```
print"xxxxx"
at 7,4
print"xxxxxxx"
get
```

The program listed above ('pro1x7') deals with something that is often very useful, arranging text or numbers into columns. This uses the keyword AT, which positions the cursor at some specified part of the screen and is in this example followed by a PRINT line (which must be a separate line). AT needs to be followed by two numbers. Of these, the first number is a 'column number', measuring position across the screen from the left-hand side, and the second number is a 'line number', measuring the lines down from the top of the screen. The column numbers range from 1 (left-hand side) to 20 (right-hand side) and the line numbers range from 1 (top of screen) to 4 (bottom of screen), with the numbers separated by a comma. Using AT followed by a PRINT line allows you to print at any place on the screen, and allows you very considerable control over printing that was lacking in a lot of versions of OPL in the past. If you try to place anything with AT that would be off the screen then off the screen it goes, and you will see nothing to help you find out what has happened. You must therefore take some care to use the correct range of numbers along with AT.

Oh, yes, how did I position the letter 'X' at the centre of the first line in the program? It's simple enough, because if the position numbers are 1 to 20, then 10 is as close as we can get to the centre. If you want a word printed centred you have to count up the number of characters that it contains. By characters, I mean letters, digits, spaces and punctuation marks. You then subtract this from 22 (you might find 21 more suitable) and divide the result by two. Take the whole number part of the answer – forget about any half left over – and this is then the correct number to use as the first figure in AT. Later, you'll see that we can use letters in place of numbers in the AT and other instructions. This allows us to centre words without all the fuss of counting letters - but that's more advanced programming than we should be thinking about at this point. Right now, you might like to think about how you

could display the words 'MY ADDRESS' centred on the screen, with your address shown neatly printed lower down the screen. When you have achieved this, you will have learned quite a lot about the use of AT.

Procedures

What in any other variety of BASIC is called a program is referred to as a 'procedure' in OPL. The reasons for this will be much clearer later on, but one aspect of a procedure is that it can be 'called' by using its name. In other words, if you put inside a program (a procedure) the name of another procedure, then the second procedure will be run when that line is executed. Suppose, for example, that we take any two of the sample programs that have been used in this Chapter. Each of them is a procedure, and in OPL terms this means that each of them starts with a filename followed by a colon on the first line – there must be nothing else on the first line. To show just what this business of procedures implies, alter the procedure for program 'pro1x5:' so that it reads:

```
pro1x5:
print"OPL - ";
print"the way ";
print"to program"
get

pro1x7:
get
```

assuming that these examples have been recorded using the filenames of 'pro1x5' and 'pro1x7' respectively. Remember that the first five lines in this example will already exist, and what follows it is the name of another procedure (you must type in the colon this time) and another GET line.

Now when you Translate this and Run it, the action of the first procedure will be carried out, and when you press any key (because of the first GET) then you will see the action of the second procedure which will show on the screen because of the second GET. All programs in OPL are of this procedure type, allowing one procedure to run another. Later we shall see the immense advantages of using this method when we look at program design and how data can be passed from one procedure to another.

2 : Variables

LZ

Assignment

So far, our computing has been confined to printing numbers and words on the screen, using program lines containing the PRINT keyword. That's covered two of the main aims of computing, processing and output, but we have to look now at some of the actions that go on before anything is printed. One of these is called assignment. Take a look at the program listed below ('pro2x1):

```
pro2x1:
local x
x=23
print"2*x is ";2*x
x=5
print"x is now ";x
print"2*x is ";2*x
pause 100
```

Type it in, run it, and contrast what you see on the screen with what appears in the program. The first line that is printed gives the text on the screen:

```
2*x is 46
```

but the numbers 23 and 46 don't appear in the PRINT line of this procedure. This is because of the way we have used the letter X as a kind of code for the number 23. The official name for this type of code is a 'variable name'. As with keywords, the case of variable names is unimportant, so you can enter 'x' or 'X' in your program with the same results. Again, like keywords, they are shown in this text in upper case.

Because of the way that OPL works with procedures, we have to start by specifying that X is 'local', which means that various values can be used for X within this procedure. If you do not use this LOCAL X line, then the procedure will Translate without trouble, but will not run correctly. The reason is that when you do not use LOCAL X, the procedure expects to find X being used in another procedure which has been running and which has called this one. We have not encountered this so far because none of the sample procedures in Chapter One required this type of use.

The second line of the procedure (ignoring the name line) assigns the variable name X, giving it the value of 23. This means that wherever we use X, not enclosed by quotes, the computer will operate with the number 23 instead. Since X is a single character and 23 has two digits, that's a saving of space. It would have been an even greater saving if we had assigned X differently, perhaps as X=2174.3256, for example. The second line then proves that X is taken to be 23, because wherever X appears, not between quotes, 23 is printed, and the 'expression' 2*X is printed as 46. We're not stuck with X as representing 23 for ever, though. The next line assigns X as being five, and the following lines prove that this change has been made.

That's why we call X a 'variable' – we can vary whatever it is we want it to represent. Until we do change it, though, X stays assigned. This very useful way to handle numbers in code form can use a 'name' which must start with a letter. You can add to that letter other letters or digits, but not spaces or punctuation marks, so that N, name, and N504 are all names that you can use for number variables, and each can be assigned to a different number.

Just to make it even more useful, you can use similar 'names' to represent words and phrases also. The difference is that you have to add a dollar sign ($) to the variable name. If N is a variable name for a number, then N$ (pronounced en-string or en-dollar) is a variable name for a word or phrase. The computer treats these two, N and N$, as being entirely separate and different. The name for a number must not end with the $ sign.

There are, as you might expect, some rules to observe. You can pick names which use more than one letter or digit, up to eight characters long – but remember that each character takes up space in the memory and might have to be typed many times over. If you can work with single-letter variables you will make the effort and the strain on the memory both considerably less. You must avoid using any name for a variable that is the name of a keyword, like AT, and this is another good reason for using single-letter 'names'.

Types of Numbers

Most versions of BASIC force you to decide what type of numbers you want to represent with each variable name, and OPL follows the same pattern. The two main types of numbers are 'integer' and 'float' (or 'floating point'). An integer number is a whole number, which can be positive or negative, but which contains no fractions. The range of an integer in OPL is between -32768 and +32767. A float, by contrast, can be any number in ordinary form, whole or fractional, positive or negative, and with a huge range of values whose limits you are most unlikely to encounter.

The reason for using different types of number variables is that integers can be stored in less space, dealt with more quickly, and are always precise. The way that a float is stored in the machine can allow errors to build up, and this can cause difficulties in financial programs, among others.

An assignment to an ordinary variable, such as X, COST, Z2 and so on, will be carried out automatically to a float type of number. For example, if you make the assignment ITEM=32, then the word ITEM is used as a variable and the number 32 is assigned as a float, meaning that it is stored in float form, even though it is an integer number. If, on the other hand, you assigned SUBTOT%=1452, then this name would be used as a integer variable for the integer number, because of the use of the % sign at the end of the name. OPL uses two distinguishing signs following variable names, $ for strings, as we saw earlier and % for integers.

Keeping to numbers for the moment, the use of a variable name unmarked by % will ensure that the variable uses the float type of storage. This means that numbers will appear correct to twelve figures so that 100.0/3 is printed as 33.3333333333 and 10000.0/3 is printed as 3333.33333333, a total of twelve figures for each.

- One point you need to be careful about is a PRINT line which results in a number being printed, because there is nothing in a PRINT line that determines whether a number is an integer or float. The rule is that if a number contains no decimal point and is within the size limits, it is an integer, and this can result in some odd results. For example, using:

  ```
  PRINT 100/3
  ```

  gives the result '33' because this is the nearest integer, neglecting fractions. Using:

  ```
  PRINT 100.0/3
  ```

  or alternatively:

  ```
  PRINT 100/3.0
  ```

  where either number has a decimal point, will give the float result of 33.3333333333 as you would expect.

The lack of precision of a float is usually concealed because when a number is printed, it can be rounded up or down to its correct value. An integer variable, by contrast, is always perfectly precise, but its range is limited and no fractions are permitted.

Use integers for as many of your number applications as possible, because they take up less memory space and can be processed quicker. Floats should be reserved for essential data that contains fractions or needs a large range of numbers.

Working with Number Variables

There's nothing particularly difficult or new about working with

number variables, though the idea might be a novelty to you if you never encountered algebra at school. Using variable names, you can type instructions like X*V, meaning that whatever number is assigned to variable X will be multiplied by the number that has been assigned to variable V. The value of being able to carry out arithmetic on variables in this way is that it allows you to work with any numbers. For example, if you put into a program the instruction statement PRINT 40.5 * 0.15, then this will print 6.075 every time you run the program.

If, however, your instruction is PRINT X*V, then what gets printed depends on what has been assigned to X and to V, and these assignments can be changed during the program. You can, for example, make the computer carry out this action many times, using a whole list of numbers. Another option that we'll look at shortly, is to make the computer assign a variable with a number that you type while the program is running. Once again, this will be a number that was not put into the program at the time when the program was written, and that's the whole point about programming. You can even make the program read hundreds of numbers from memory and work on these.

- One point to watch is that you cannot use as variable names the names of arithmetical calculator routines, like SUM, MAX or MIN. Such names are rejected when you Translate the program.

Simple arithmetic with variables, then, is typed much as you type simple arithmetic with numbers, using the signs '+', '–', '*' and '/'. This means that you can have lines containing items such as:

```
S=F+X
PROFIT=GROSS - EXPENSES
VAT=price*rate
```

or:

```
METRES=MILLS/1000.0
```

The result of such pieces of arithmetic can be printed or, as shown here, assigned to some variable that will be used later. One complication occurs, however, when the same variable name is used twice. For example, what do we mean by the assignment: S=S+ITEM? By the ordinary rules of numbers, this would just be silly, and its meaning hinges on a different use of the '=' sign. Used in this way, the '=' sign means 'becomes', so that the action of S=S+ITEM means that you add together the value of S and of ITEM, and make this the new value of S.

No language other than BASIC uses the '=' sign for both purposes in this way, and it's something that you just have to get used to. Wherever the same variable name appears on each side of the '=' sign, then the '=' sign means 'becomes', so that statements like A=A*B or C=C–X or D=D/5 are all examples of this use of the equality sign.

String Section

The following listing ('pro2x2') illustrates 'string variables', meaning the use of variable names for words and phrases.

```
pro2x2:
local l$(20),f$(20),w$(20)
cls
l$="OPL Basic"
f$="The excellent"
w$="program language"
print f$,l$,w$
print"This uses ";l$
print f$,w$;" in action"
get
```

This causes some complications that are not present when we use numbers. When you work with number variables, the computer can set aside a fixed amount of memory for each number, two units (bytes) for an integer, and four for a floating point number.

A string, however, can consist of any number of characters, and though some varieties of BASIC can cope with this, OPL demands that the maximum number of characters should be specified for each string variable. In the LOCAL line for this example, then, each string variable is declared as having up to 20 characters, using the form:

```
L$(20),F$(20),W$(20)
```

to show the names and lengths. In this example, 20 has been taken as a convenient length since this is the maximum number of characters that will fit across the screen width.

The next three lines carry out the assignment operations, and the other lines show how these variable names can be used. Notice that you can mix a variable name, which doesn't need quotes around it, with ordinary text, which must be surrounded by quotes.

You have to be careful when you mix these two, because otherwise it's easy to run words together. Note in the last three lines how spaces have been left between words this time, using the comma in place of a semicolon. Another way of tacking strings together, incidentally, is to use a + sign to connect the variable names, such as:

```
PRINT L$+F$+W$
```

but if you do this you must either have suitable spaces in the strings, or alter the command to read:

```
PRINT L$+" "+F$+" "+W$
```

with the space represented by pressing the space key between the quote marks.

Strings and Things

Because the name of a string variable is marked by the use of the $ sign, a variable like A$ is not confused with a number variable like A. We can, in fact, use both in the same program knowing that the computer at least will not be confused.

```
pro2x3:
local a,b,a$(2),b$(2)
a=2
b=3
a$="2"
b$="3"
print a,b
print a$,b$
print"a*b is",a*b
print"a$*b$ is impossible"
get
```

The listing above ('pro2x3') illustrates that the difference is a bit more than skin deep, though. The first two lines assign number variables A and B, and string variables A$ and B$. When these variables are printed further on, you can't tell the difference between A and A$ or between B and B$. The difference appears, however, when the computer attempts to carry out arithmetic. It can multiply two number variables because numbers can be multiplied, but it can't multiply string variables, whether these represent numbers or not. You can multiply 2 by 3, but you can't multiply "2 LABURNUM WAY" by "3 ACACIA AVENUE". The computer therefore refuses to carry out multiplication, division or any other arithmetic operation on strings – the exception being 'addition' (more about that later).

Attempting to do a forbidden operation causes a message:

```
ERROR TYPE MISMATCH
```

to be delivered when you try to Translate, meaning that you have tried to do on strings what can only be done on numbers – you are trying to use the wrong type of variable. Later on, we'll see that there are operations that we can carry out on strings that we can't carry out on numbers, and attempts to do these operations on numbers will also cause an error message. The difference is an important one. The computer stores numbers in a way that is quite different from the way it stores strings.

The different methods are intended to make the use of arithmetic simple for number variables (for the computer, that is), and to make other operations simple for strings. Let's face it, it's only a machine!

There is one operation that looks like arithmetic that can be carried out on strings. It uses the + sign, but it isn't addition in the sense of adding numbers; it is called 'concatenation'. Concatenation is a very useful way of obtaining strings which otherwise would need rather a lot of

typing, and of forcing one string to be tacked on to the end of the other. Take a look at the following listing ('pro2x4'):

```
pro2x4:
local a$(5),b$(6),c$(9)
a$="*****"
b$="%%%%%%"
c$="OPL Basic"
print b$+b$+b$
print a$+c$+a$
print b$+b$+b$
get
```

This defines strings A$ and B$ as characters which can be used as 'frames' around a title. The title is defined as 'OPL Basic'. The top and bottom of the frame are produced by concatenating three copies of B$. In addition, the sides of the frame and the title are printed as a concatenated string: A$+C$+A$.

Getting Some In

So far, everything that has been printed on the screen by a program has had to be placed in the program before it is Run, by direct assignment. Our programs have just consisted of a bit of processing and some output, but with no input apart from whatever was placed in the program. We don't have to be stuck with restrictions like this, however, because the computer allows us a way of putting information, either numbers or names, into a program while it is running. A step of this type is called an 'input' and the OPL instruction word that is used to cause this to happen is also INPUT.

The following listing illustrates this with a program that prints your age. Now I don't know your age, so I can't put it into the program beforehand.

```
pro2x5:
local b%,y%
print"Year of birth"
input b%
print"Current year"
input y%
```

```
        print"You are",
        print y%-b%,
        print"this year"
        get
```

What happens when you run this program ('pro2x5') is that the words:

```
        Year of birth
```

are printed on the screen, and below this you will see a dash '–'; the Organiser has no question mark on its keyboard and we'll look at how to get such symbols later. The question mark is used for other purposes, as we'll see later. The computer is now waiting for you to type something, and then press EXE. Until the EXE key is pressed, the program will hang up waiting for you. If you're honest, you will type your year of birth and then press EXE. When you press EXE, your year of birth is assigned to the variable B%. The program can then continue, so that the question 'Current year' is then asked. Once again, you answer by typing the year and pressing EXE, and the reply is assigned to variable Y%. In fact, this year could be obtained from the computer's built-in calendar, but that's a refinement that will have to wait for now.

The program then prints the age you will be on your birthday this year, assuming that you answered correctly. You could, of course, have answered 1392 or 1745 or anything else that you pleased for either of these years. The computer has no way of knowing *in this program* that either of these is not your true year of birth or the current year. Don't listen to the nutters who tell you that computers know everything!

Now that you can type something that can be assigned to a variable, and then use the variable later, you can use all three of the main computing actions. Could you now design a program that asked for your annual income, and assigned it, and then asked for your taxcode, and assigned that? Could you then arrange it so that it then cleared the screen, and printed your income after tax (knowing that the amount of tax is 'taxrate*(income–10*taxcode)')? You now know all of the commands that are needed.

INPUT can be used with an integer variable as illustrated here, with a floating point variable, or with a string variable (so that you can enter names, for example). The only difference on the form of the command is the type of variable name that follows it. There are, however, differences in the way that the computer accepts an input. If the INPUT command uses a string variable, like INPUT A$, then anything that you type is acceptable. If a number variable is used, like INPUT A or INPUT A%, then what you type must be a number, and for INPUT A% it has to be a number that is an integer in the correct range of –32768 to +32767. If you enter characters when you are asked for a number, the computer will print a question mark so that you can try again with a sensible entry this time.

- When you have an INPUT line that calls for a number, the number-keys of the Organiser will provide numbers with no need to press the SHIFT button.

Single Key Reply

So far, we have been putting in replies with the use of INPUT, which means typing and then pressing EXE. You could use this for single-letter replies (Y or N), and this has the advantage of giving you time for second thoughts, because you can delete what you have typed and type a new letter before you press EXE.

For snappier replies, however, there is an alternative in the form of GET$. GET$ is an instruction that carries out a check of the keyboard to find if a key is pressed and repeats this until a key is pressed – so far we have used the similar instruction GET as a way of making a program wait for a key to be pressed. The syntax of GET$ is always of the form: K$=GET$, so that the string variable K$ carries whatever has been assigned to it by GET$, and if you are looking for a single-key reply then you should make K$ a one-character variable, using a line such as:

```
        LOCAL K$(1)
```

at the start of the procedure. The following listing ('pro2x6') shows such a GET$ being used for a Y or N reply – though in this simple example, any key could be pressed – we'll look later at how a reply can be tested.

```
pro2x6:
local k$(1)
print"reply y or n"
k$=get$
print"Thats",
print k$
get
```

The GET$ instruction will produce a string quantity when any key is pressed, so we must assign GET$ to a string variable such as K$. In this way, when a key is pressed, the quantity that it represents will be assigned to K$, and we can then test this string as we want.

Operators

An operator is a symbol for a fundamental mathematical operation. If that definition looks intimidating, don't worry, because the main operators that you are likely to use are the familiar signs '*', '/', '+' and '–' which carry out the fundamental operations of multiply, divide, add and subtract. Each of these is an operator that requires two numbers to work on, and in some books you will find them called 'binary operators'. The numbers (or in some cases strings) that operators work on are called operands. The operators of OPL are of four kinds, classed as arithmetic, string, relational and logical. The first group is composed of the four symbols that we are familiar with, plus the exponentiation action of raising a number to a power (using **), and the % sign (more about that later). The only string operator is the +, which will concatenate strings (join them together). In addition to these familiar tasks, you can use the '+' and '–' signs as 'unary' operators, meaning that they can be used on a single number. You can, for example, write things like +2.54 or –3.6, and these are examples of making use of the '+' and '–' operators in a unary way.

The relational (or comparison) operators are the set of signs that show relationships, rather than producing some answer. The main three

signs in the group are '=' (equal to), '<' (less than) and '>' (greater than), which compare the size of numbers and the ASCII codes of string characters. These signs can be combined, so that '>=' means greater than or equal to, '<=' means less than or equal to, and '<>' means not equal to. These are read left to right, so that 'A>B' means 'A greater than B', and 'X<Y' means 'X less than Y'.

Finally in this list, the logic operators are the words AND, OR and NOT, sometimes called the Boolean operators in honour of the mathematical genius George Boole whose work laid the foundations of computing science in the 1840's. The action of a logic operator is to return TRUE or FALSE when it is used to test a relationship. The NOT operator is unary, and it gives a TRUE result if what it precedes is NOT TRUE. The machine expresses TRUE as the number –1 and FALSE as 0. If you are not accustomed to this, it can look very confusing, and a few examples will help. The secret is to work in terms of TRUE and FALSE only, and to start with any term that is enclosed in brackets. For example, what do you expect from the line:

```
PRINT NOT (2>1)
```

when this runs? The answer is worked out by looking first at 2>1, which is TRUE. NOT TRUE is FALSE, so the answer must be the code for FALSE, which is 0. Try another one:

```
PRINT NOT("B">"A")
```

Note – the quotes are important.

When we compare strings, alphabetical order is treated like numerical order, so that "B"> "A" is TRUE, and NOT TRUE gives FALSE, 0. By the same token, NOT ("A"> "B") gives –1, TRUE. Using PRINT NOT(0) will give –1, since NOT FALSE must be TRUE, and equally obviously, NOT(–1) gives 0. If your nerves are up to it, try PRINT NOT(7) and see if you can explain the result. If it's baffling, please turn to Appendix B. In the normal course of programming, you should not have to be worried too much by this kind of thing, but it's as well to know, because it can sometimes make easy meat of what appears to be a difficult piece of programming.

The other logic operators, AND and OR each need two quantities to work on. These quantities can be number comparisons or string comparisons, and the important point once again is that each side of the AND or OR word should be something that can be resolved to TRUE or FALSE. For example, if we have:

```
PRINT (7>3)AND(5>2)
```

then we can expect the result –1. Why? Working out the items in the brackets we have 7>3 is TRUE and 5>2 is TRUE, so TRUE AND TRUE = TRUE. The Law of AND is that the result is TRUE only if the items that are connected are also both TRUE. If one item is FALSE, the result is also FALSE. On that basis, then you would expect the result of:

```
PRINT (5>6)AND(7>4)
```

to be 0, as it is because one term is FALSE. The OR operator will give TRUE if any one item is TRUE, no matter whether the other is TRUE or FALSE. Only if both items are FALSE will the result of OR be FALSE. Table 2.1 summarises the actions of AND and OR. Remember that you would normally be using these operators with variable names rather than numbers or letters.

A	**B**	**A AND B**
False	False	False
False	True	False
True	False	False
True	True	True

A	**B**	**A OR B**
False	False	False
False	True	True
True	False	True
True	True	True

Table 2.1. The actions of the AND and OR comparisons on two items which can be true or false.

Expressions

An expression is a set of operators and operands that provides a number or a string result. A very simple expression is 1+2, but we usually reserve the term for lines that make use of variable names, and in which more than one operation may be carried out. A familiar pair of expressions are the incrementing expression, X=X+1 and the decrementing expression X=X-1. Remember that the '=' sign in BASIC is used here to mean 'becomes' rather than 'equals'. The expression X=X+1 therefore means that the value of X is increased by one, and X=X-1 means that the value of X is decreased by one.

In general, the use of more complicated expressions is something that often proves baffling to a computer user who has no experience of mathematics. This needn't be so, because expressions, like anything else in computing, follow precise rules, and once you know what the rules are it's not difficult to apply them. The most important rules concern 'precedence' of operators, and once you know about precedence, it's not difficult to find what an expression does. Making up an expression for yourself is another matter, and only practice can help there.

Top priority:	** (exponentiation, such as 5**2=25)
	NOT (coded as –)
	* and /
	+ and -
	= > < <> >= <=
Lowest priority:	AND OR

Table 2.2. Precedence of operators. If you have more than one arithmetic operation in a line, the actions will take place in this order. Actions of equal precedence will be executed in a left-to-right order.

Table 2.2 shows the order of precedence. What this means is that if you have more than one operation in an expression, the operation(s) with higher precedence are carried out first. If there is no clear precedence, then the order in the expression is simply left to right. For example, if you have the expression:

```
PRINT 5+4*3-6/2
```

what do you expect? If everything obeyed a left-to-right order only, the result would be got from 5+4=9, 9*3=27, 27-6=21 and 21/2=10.5 (Note that the result printed would be 10 – the fraction would be lost because the expression is an integer expression). It's not like this, though. Because multiplication and division have higher precedence than addition and subtraction, the 4*3=12 and the 6/2=3 are worked out first. Having done that, all that is left is of equal precedence, and we get 5+12=17 and 17-3=14, which is the answer that the computer will give you.

Remember that in a program, all or most of the quantities would be variables, and to find the numerical answer you would have to find what numbers were assigned to the variables at the time of evaluating the expression. You might, for example, be working with something like Y=K+B*X**N. The X**N action is carried out first, since the raise-to-a-power action (exponentiation) has highest precedence, and then the result of this is multiplied by B.

Finally, the value of K is added. Precedence rules, O.K.?

One important point to remember is that brackets take precedence over everything else. For example, if you have an expression which boils down to 5*(4+3), then this is not the same as 5*4+3 (which is 23), it actually gives 35, because whatever is inside the brackets is worked out first, giving seven in this case.

When there are several sets of nested brackets, meaning brackets inside other brackets, then whatever is innermost has highest precedence. For example:

```
5*(4+(8-6/2))
```

gives 45, because the innermost bracket gives five, adding this to the four in the next layer of brackets gives nine, and the result is 5*9.

For some reason, however, it all looks much more fearsome when used with variables, particularly when there are actions like NUM$ and VAL (see later) involved as well.

Translating Formulae

The earliest computer programming languages were for scientific and engineering use, and translating formulae so that the computer could deal with them was a very important feature.

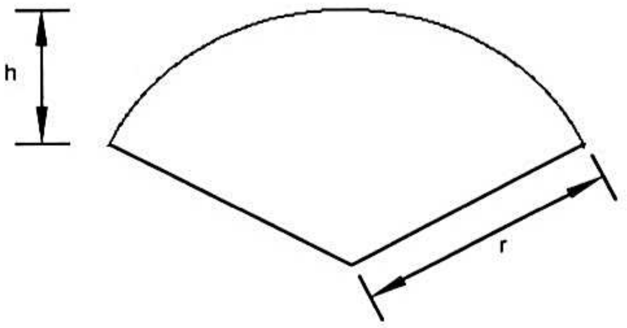
It was so important, in fact, that one of the main languages in the early days was called FORTRAN, an abbreviation of FORmula TRANslation. FORTRAN is still used, and the BASIC language which is used by all microcomputers is based very considerably on the ideas and methods of FORTRAN.

For this reason, BASIC is a language not to be despised if your needs or interests are in programming of this type. A lot of languages that are more highly regarded either by academics or for business use are inferior to BASIC when it comes to working with formulae, and also, incidentally, for dealing with strings and disc files. OPL is definitely one of the forms of BASIC that retains this tradition of dealing well with formulae.

If you haven't had some practice, however, it's not always straightforward to convert a formula written in a reference book into the form of an OPL expression. For one thing, you have to remember that the order in which the terms of the formula are written will not usually be the order in which you want them evaluated, so that you must either change the order or make use of brackets to obtain the correct expression.

Examples help here, but one person's example is another's confusion, so please bear with me if the formulae that you want to use are not shown here. Remember that you don't have to derive the formulae for yourself for most purposes, you simply take them from a reference book. You need to know, of course, what variable values have to be supplied, and what the formula does, and you also need to know any limitations, but in general this is all. You get into a different league when you start to generate your own formulae!

We'll take as a first example the formula for the volume of a sector of a sphere, shown in Figure 2.1(a). Now, like many formulae, this uses no sign for multiplication. Quantities that are printed together are intended to be multiplied, so that the formula requires you to multiply two by pi by the value of r(squared) by h, and then divide the answer by three. Note that this uses pi, whose value is supplied as this variable name in OPL.



a) Formula is: $v=2\pi r^2h/3$

b) In OPL: v=r**2*2*PI*h/3

Figure 2.1. Converting a formula (a) into the form of OPL BASIC (b).

Now in the expression, there is one power taken, and this action will have precedence no matter where we put it. It makes sense, in any case, to start with this item, getting the value of 'r' squared. Suppose that we use variable names 'R' and 'H', then the expression that is shown in Figure 2.1(b) is the OPL expression for evaluating the formula. The variable V is used for volume, and the main point to note is that we have to insert the multiplication signs '*' and the division sign '/' that OPL demands.

Because all the operations apart from finding the square are of equal precedence, we can write the rest of the expression in left-to-right order, and be reasonably confident. In all cases, however, if you are in any doubt, try a few examples with simple numbers and check that you get what you expect. In this example, the answers will always have more places of decimals that you would really want to use, and we'll look at how to round them off later.

The real problems come when the formula is not in the form that you want. If you have a smattering of algebra (ie, if you are over the age of 40) then you may be able to rearrange the formula to suit. Old textbooks of elementary algebra (like the Hall & Knight of blessed memory) deal with this important topic much better than modern textbooks which seem to require you to discover it all for yourself, or assume that no-one needs such things any more.

Functions

There are many quantities that we need to calculate which cannot be dealt with by an operator, or even by a reasonably simple expression. Quantities such as trigonometrical ratios, square roots, hexadecimal equivalents and so on are dealt with by the use of functions. A function of a number is a quantity that is obtained by the use of various actions on the number. The number (or more likely, variable) is called the 'argument' of the function, and for many functions has to be enclosed in brackets.

In the computing sense, functions can also include actions on strings, and the main thing that they have in common is that the function uses a statement word (not a symbol, as an operator uses) and that it needs an operand or argument, which can be a number or a string depending on the type of function. The main number functions of OPL are listed in Table 2.3, along with their effects.

Of this list, the trigonometric functions merit particular attention, because they cause a lot of trouble to programmers who are working with trigonometrical formulae for surveying and similar calculations. The functions that are most used are SIN, COS and TAN, all of which are provided in OPL. What you need to watch, however, is that each of these functions needs an argument which is an angle in radians.

Now if you are working with angles in degrees you will need to convert from degrees to radians, and there is a conversion function built into OPL. By using an angle D in degrees, you can convert by using R=RAD(D), so that R will hold the angle converted into radians.

The reverse problem occurs when you need to use the inverse trigonometric functions, such as ACOS, ASIN and ATAN. Each of these finds the angle (in radians) whose cosine, sine or tangent (respectively) has the value which is used as the argument. This is called an inverse function because it finds the angle rather than the function of an angle. ATAN is a function which you don't need all that often. A lot of work with trigonometry calls for the inverse SIN (ASIN) and inverse COS (ACOS) rather than the inverse TAN (ATAN). The angle that you obtain when you use these functions is in radians, and can be converted to degrees by using the DEG function. This can be combined with the trigonometrical function itself, such as in:

```
D=DEG(ATAN(1))
```

The other function which can cause problems is the LOG function. When you use a statement such as A=LOG(X) then what is assigned to A is the familiar base 10 logarithm – OPL is one of the select number of varieties of BASIC that uses LOG to mean the base10 logarithm. The natural log of X makes use of LN, which agrees with the use of LOG and LN in science and engineering. Schools nowadays don't teach about logarithms, because subject advisors and teachers don't seem to know to what extent logarithms and logarithmic functions are used in engineering and scientific work – it's yet another reason for firms setting their own entrance exams.

Function	**Application**
ABS(x)	Strips negative sign from floating-point number x
ACOS(x)	Gives angle whose cosine is x
ASIN(x)	Gives angle whose sine is x
ATAN(x)	Gives angle whose tangent is x
COS(x)	Gives value of cosine of angle x radians
DEG(x)	Converts angle x radians into degrees
EXP(x)	Calculates e^x, used as inverse for LN
FIX$(x$)	Converts number to string of specified format
FLT(x)	Converts integer to floating point number
GEN$	Convert number to string for any number.
HEX$(x)	Converts integer into string of hexadecimal digits
IABS(x)	Strips negative sign from integer number x
INT(x)	Rounds to nearest lower whole number, returns an integer
INTF(x)	Rounds as INT, but returns a floating-point number
LN(x)	Gives natural logarithm of a number x
LOG(x)	Gives logarithm to base 10 of x
NUM$	Gives string form of floating-point number
PI	Gives value of π
RAD(x)	Convert from x degrees to radians
RANDOMIZE	Set sequence of 'random' numbers
RND(x)	Gives 'random' fraction
SCI$	Convert number to string, using scientific format
SIN(x)	Gives value of sine of angle x in radians
SQR(x)	Gives square root of x. x must be positive
TAN(x)	Returns value of tangent of angle x in radians

Table 2.3. Most functions act on a number or string, which is referred to as the argument of the function; a few functions require more than one argument. Most functions require the argument(s) to be placed between brackets. In the following list, x is used for a number argument (usually floating point, but in some cases an integer) and x$ for a string.

```
pro2x7:
local b,c,a
print"first side",
input b
print "second side",
input c
print "angle between",
input a
print"Area is",
print .5*b*c*sin(rad(a))
get
```

The listing above ('pro2x7') gives an example of the use of these trigonometrical functions to find the area of a triangle which is not right-angled, a common surveying application. The angle is entered in degrees and is converted to radians as part of the expression. The printed statements are rather curt, and if you wanted this program to be used by someone with little experience you would have to provide more lines of print for each entry, detailing the units of measurements and so on. The formula assumes that the units are consistent, so that if the length is measured in metres and the angle in degrees, then the area will be in units of square metres.

Another function that you often need for a surprising number of applications is RND, which generates a 'pseudo-random' number. Games, prize draws, and many statistical actions (like the Monte-Carlo method) require numbers that are taken at random. Now a computer works to fixed rules, and any number that is obtained from a formula cannot be totally random, but by using a sufficiently complicated formula you can get something that is almost a random number. To be more precise, it's a random fraction with a value which will always be more than zero and less than one, and the following program illustrates how this fraction is obtained and how it can be multiplied by numbers to get random number sequences in whatever range you want. You should get different fractions and a different whole number each time you use this.

```
pro2x8:
print"Look at some fractions"
print"taken at random."
pause 20
print rnd
print rnd
print rnd
print rnd
pause 40
cls
print"Now a number "
print"between one and 100"
print int(100*rnd)+1
get
```

The important point in this example ('pro2x8') is to see how a random fraction, which can be anything from 0.00000000001 to 0.99999999999 can be converted to a range of whole numbers. Suppose we want numbers in the range 1 to 100. Now multiplying the extremes of the

fractional range by 100 gives 0.000000001 to 99.999999999. Taking the integer part of this with INT gives the range 0 to 99. Adding one gives the range 1 to 100. The method of converting then is to multiply RND by your upper range number, take the INT, and then add one.

Print Appearance

One problem that the example above makes obvious is that it can be very awkward to get lines of print looking neat and well-arranged in their 20-character sets. OPL provides some help in the shape of DISP and VIEW, as illustrated in the following short listing ('pro2x9'):

```
pro2x9:
local a%,a$(80)
a$="This is a demonstration of an
alternative to the PRINT command."
a%=disp(1,a$)
cls
a%=view(2,a$)
```

Either of these commands allows a string to be displayed scrolling in one line from right to left, and controlled by the cursor keys, so that the right and left cursor keys can be used to halt, reverse or accelerate the speed of the scrolling. For DISP only, the up and down cursor keys put the start of the message on to the screen line. The message keeps scrolling, wrapping around on itself so as to scroll continuously, until a key other than a cursor key is pressed.

The differences between DISP and VIEW are not particularly important at this point, but VIEW has several advantages for use with information strings like this (DISP is designed for reading files). To start with, VIEW allows you to specify the line number on the screen (one,. two, three or four), using this as the number in the brackets along with the string variable name. The other feature is that the variable A% (you can, of course, use any variable name you want) will contain a number-code for the key that you press to stop the scrolling of the display, so that the GET action is combined with a PRINT action and with scrolling in this useful command.

Precision of Numbers

One problem that turns up time and time again in computer work is the precision of numbers. For some reason, computers are always thought of as being associated with mathematics, and people believe that computers can carry out arithmetic much more precisely than your average £3.50 pocket calculator. Don't you believe it! This is something that causes more trouble than anything else in computing (apart from printing a pound sign), and it arises because of the way that computers store numbers in the memory. OPL allows for numbers to be stored in two main ways, called integer and float. Up to now, we have made use of numbers without saying much about them, but now the time has come to explain more, starting with integers.

An integer means a whole number, but for the purposes of OPL, it has a more restricted meaning of a whole number whose range is from –32768 to +32767 only. Now a number in this range can be stored in two of the memory units that we call bytes, and its value will always be precise.

If you want to specify a variable as an integer, you use the % sign to mark it out, such as A% or SM%. Because an integer requires only two bytes for storage, the use of integers will increase the running-speed of a program, and we shall see shortly that this can be turned to a considerable advantage.

If you can perform all arithmetic with integers, it will be fast and, with one exception, precise. The exception is division, because an integer number cannot be assigned with a fraction. If, for example, you assign A%=B%/C%, with B%=5 and C%=3, then A% will be 1, not 1.66666666667, which is what you would get from PRINT 5.0/3.0.

You should use integer variables when:

1) The number range that you are likely to work with is comparatively small and does not require fractions.

2) The number variable is used many times, particularly for a constant value.

3) The number variable is used in expressions, particularly in loops (see Chapter Three).

By keeping to these rules, you can make your programs run faster and take up less of the memory. When you first start to program in OPL, these points aren't exactly the most pressing ones for you, but later you'll need to know and make use of these points. From now on, if it's going to be an advantage to work with integers, the examples in this book will show them in use.

Floating Point Numbers

When you need to work with floats or reals (short for floating point or real numbers), meaning numbers which can be positive or negative, whole or fractional, and with a much greater range of size, precision then becomes a problem.

In a lot of applications, we use real numbers in standard form, which means that a number such as 52400 would be written as 5.24E4, meaning 5.24 x 104. When this is done, numbers are often approximated, so that the number 52417 might also be written as 5.24E4 on the grounds that the extra 27 represented only a small fraction of the whole number. To put it another way, the precision of the number is sacrificed so that it can be written with only three digits before the E sign. This part of the number is called the mantissa, and the part which follows the E is called the exponent.

When a real number is stored in the computer, it is also stored in mantissa-exponent form, but with both in binary numbers, using the digits '1' and '0' only. The mantissa is a binary fraction, stored in seven bytes, and the exponent is a single byte integer. This is a compact way of storing numbers, but it does mean that there will always be some approximations.

```
pro2x10:
local a,b,c
a=1.0/11
b=7.0/11
c=6.0/11
```

```
        print"a is ";a
        print"b is ";b
        print"c is ";c
        get
        print"b-c is ";b-c
        get
```

The listing above ('pro2x10') illustrates this, using assignments to three variables. Running this shows that the two numbers, A and B–C do not give precisely the same result on the screen, appearing in the form 9.0909090909091 for A and 9.090909090909 for B-C, so that if we were to test for these numbers being equal the test would fail because they are not precisely equal. The answer is to use another form of comparison. Instead of comparing, for example A with B-C in this example, we could compare the quantities:

```
        INT(1000*A)
```

and:

```
        INT(1000*(B-C))
```

which would be precisely equal. This is because the INT action removes the tail end of the number which causes the problems. For example, 1000*A is 9090.9090909091, and INT(1000*A) is 9090, which is the same figure as will be obtained from INT(1000*(B-C)).

The sensible use of the INT form of expression, therefore, can solve a lot of problems that otherwise can result from the use of floats. If, for example, you are writing an accounts program in which two quantities are expected to balance, then rounding will be essential. This is because you will be working with floating-point numbers which include two places of decimals, and because of storage errors, there may be small fractional errors. If you are displaying your results only, these fractions are of no importance and do not appear, but they can cause any comparison to be incorrect. Two columns of figures, for example, which should give the same amount, will not necessarily give a true answer unless you use the adjusted amounts such as INT(100*X) when comparisons have to be made. If the number that you

want to round off would be too large to be an integer (larger than 32767) then you can use the INTF functions which works like INT to remove fractions but gives a floating point number. Chapter Three deals with the comparison actions.

Number Notes

Working with numbers implies the input of numbers from the keyboard, processing, and the display of numbers on the screen. As far as input is concerned, the conventional method is the statement of the form INPUT A (float) or INPUT A% (integer). Using INPUT A%, you will get no error message if the number that you enter for an integer is a small fraction, such as 24.5, but you will not be allowed to enter an integer of incorrect range.

The number that is assigned to the variable must be an integer so that if you enter 22.56 for INPUT A%, then A% will hold the number 22, the integer part of the number. If you try to enter an out-of-range numbers such as 32769, then you will get the question mark reminder and be given the chance to try another entry. Using a number variable, incidentally, does not mean that you cannot enter any letters, because you can always enter a number in a form such as 1E3 (equal to 1000) when the entry is to a float or to an integer.

One principle that is employed by a lot of programmers is to use a string for entry, such as INPUT A$. No entry, unless it's a string of excessive length, will be rejected, and it's then easy to check what has been entered and issue messages about errors. It's also easy to convert a string form of number into number form, using VAL, dealt with later.

Finally, OPL permits you to use a set of ten variable names which need not be declared in advance, and which can be used outside an OPL procedure. These are the calculator memory stores 'm0' to 'm9'. If your OPL program makes use of these variable names they need not appear in any LOCAL or GLOBAL line, and the values that are assigned to these variables are retained after the OPL procedure is completed. You can then switch to the calculator and make use of the values stored as m0 to m9 in any further calculations you want to make.

3 : Getting Repetitive

One of the activities for which a computer is particularly well suited is repeating a set of instructions over and over again and every computer language is equipped with commands that will cause repetition. OPL is no exception to this rule, and it is equipped with more of these 'repeat' commands than is usual for other varieties of BASIC. We'll start with one of the simplest of these 'repeater' actions, and one which we should never have to use, GOTO.

GOTO means exactly what you would expect it to mean – go to another point in the program. Normally a program is carried out by executing the instructions in the order in which the lines are placed in the memory. Using GOTO can break this arrangement, so that a line or a set of lines will be carried out in the 'wrong' order, or carried out over and over again. The command word GOTO can be used along with a 'label name', and you can make the name one that reminds you of what you want to do. When a label is used, the point to which you want GOTO to take you is marked by a statement such as loopit::, a word followed by two colons, which has to be placed on a preceding line of its own.

The trouble with this sort of thing is that it's possible to create in this way a repeating 'loop' that can be very difficult to break. The manual tells you that pressing the ON/CLEAR button followed by 'Q' (for Quit) will break a loop, but this particular type of loop is very difficult to break in this way if it contains an INPUT step.

If you cannot break a loop by use of these buttons you will have to break it by disconnecting the battery of the Organiser, which means that you lose all of your stored data. The way out of the problem is to make every loop contain a test which can be used to end the loop. Such a test can be formed by using the instruction keyword IF.

IF has to be followed by a condition. You might use conditions like IFN%=20, or IF NM$="LASTONE" for this purpose. After the condition, you can use as many lines as you want for actions that will be carried out when the condition is TRUE, and this set of lines ends with the keyword ENDIF. When the condition is tested and found to be FALSE, only the program lines following ENDIF will be executed; the lines between IF and ENDIF are for use when the condition is TRUE.

```
pro3x1:
loopit::
print"Looping fills screen"
if key<>0
stop
endif
goto loopit::
```

The program listing above ('pto3x1') shows an example of a very simple loop, which is tested by checking if any key has been pressed. This is done using the function KEY, used in the form of a test: IF KEY<>0, meaning that the test is TRUE if any key has been pressed. The function KEY can also be used in the form N=KEY which makes N=0 if no key has been pressed, but assigns the number code (called ASCII code, see later) for the key if a key has been pressed. In this example, the only action line is the one which prints a phrase on the screen. The first line is labelled by using Loopit::, and the sixth line uses GOTO Loopit:: to force the computer to repeat the action endlessly. The test follows the print action, and if TRUE causes the STOP action which will end the looping. If no key is pressed, the looping continues for as long as power is supplied.

Now GOTO is a method of creating loops that we prefer not to use if anything else is available, mainly because we have to be so careful about how to get out of the loop. The main use of GOTO is not in forming loops, but in allowing a program to jump over a step or set of steps – and in OPL there are always other options, so that you should never need to use GOTO.

GOTO allows you to get to any point that you care to mark with a label name. Because of this, it's all too easy to get a GOTO wrong by placing

the name at the wrong position. This is more likely when you are working with a long program, and you can't see on the screen the line that you want to go to. Chapter Six shows one of the few applications of GOTO to form a loop when errors are being detected.

The type of loop that we demonstrated above can be created in a very different way in OPL, using one of the several 'structured' loops that OPL provides. Structured means that the loop has its start and its end marked, so that you can see at a glance which instructions are being repeated, even if the loop is a long one. We'll concentrate on structured loop methods from now on, and show next an example that uses a structured alternative to GOTO, and with rather more action.

```
pro3x2:
local n%
n%=0
do
print"OPL No.",n%
n%=n%+1
until n%=20
```

In the above example ('pro3x2') the first line declares the variable N% and the second line assigns a starting number of zero. This is not strictly necessary, because any local number variable will have a zero value at the start of a procedure, but it's a good habit to acquire in case you need a starting value that is not zero. This is then followed by the reserved word DO. The principle of the DO type of loop is that the word DO marks the start of the loop, and every instruction that you put between DO and the word UNTIL that marks the end of the loop will be repeated. The number of repetitions is controlled by placing a condition following UNTIL; this condition can be a test of a number or string variable, or something like KEY which tests for a key being pressed.

In this example, the test is for the number variable N% reaching the value of 20. A very important part of the program is the line that increases the value of N% by unity ('incrementing' N%). If this is omitted, N% can never reach 20, and the test always fails, allowing the loop to run indefinitely. To stop such a runaway loop, hold down the

ON/CLEAR key until you hear the buzzer sound, then release this key and press the Q key – you have to move from ON/CLEAR to Q very quickly.

The loop will cause the screen to print the words:

```
OPL No.
```

and the number that is held in variable N%, each time the computer goes through the actions of the loop. We call this 'each pass through the loop'. This continues until N% reaches the value of 20, since this satisfies the UNTIL test.

The important point about this type of loop is that the number of repetitions is strictly fixed by the UNTIL test – providing that any number variable that you use in the test is correctly incremented or decremented in each loop. You don't have to confine this action to single loops either. The following listing ('pro3x3') shows an example of what we call 'nested loops', meaning that one loop is contained completely inside another one.

```
pro3x3:
local n%,j%
do
print"Count is",
print n%
j%=0
  do
  j%=j%+1
  until j%>=1000
n%=n%+1
until n%>=10
```

When loops are nested in this way, we can describe the loops as being inner and outer. The outer loop starts in the second line, using variable N% which goes from 0 to 10 in value. The next line is part of this outer loop, printing the value that the counter variable N% has reached. We then create another complete loop. This must make use of a different variable name, and it must start and finish again before the end of the outer loop. We have used variable J% this time, and we have ensured

that the value of J% starts at zero each time. Otherwise, on the second and subsequent times that the computer passed through the outer loop, it would find the value of J% to be >=1000 already and so the inner loop would only run once. In this inner loop, there is nothing placed between the DO and the UNTIL parts to be carried out.

All that this loop does, then, is to waste time, making sure that there is some measurable time between the actions in the main loop. The last action of the main loop is incrementing the value of N%, and the UNTIL N% >=10 is placed in the final line. Note that the inner loop has been indented by a couple of spaces. This is not necessary but it does make the program easier to follow.

The overall effect, then, is to show a count-up on the screen, slowly enough for you to see the changes. Note that both tests have been formed using the test '>=' so that the test will fail if the number is either greater than or equal to the specified number. This avoids the possibility of a test failing because somehow a value has been skipped. Such a test is particularly important for floating point numbers when exact equality cannot be guaranteed.

Even at this stage it's possible to see how useful a DO...UNTIL loop used with a number variable can be, but there's more to come. To start with, the loops that we have looked at so far count upwards, incrementing the number variable. We don't always want this, and we can make the number variable be incremented in other steps, or decremented in value, in the course of the loop. We could, for example, use a line like:

```
N%=N%+2
```

which would cause the values of N to change in the sequence 0, 2, 4, 6, 8, and so on, or we could use a starting value of N%=12 and a loop step such as:

```
N%=N%-3
```

to make the values descend in the order 12, 9, 6, 3, 0, and to negative values if we have not tested for N%<=0.

```
pro3x4:
local n%
n%=10
do
print n%,"seconds - "
print"and counting"
pause 20
cls
n%=n%-1
until n%<=0
print"Blastoff"
pause 20
```

The listing above ('pro3x4') illustrates a loop which has a step of –1, so that the count is downwards. Variable N% starts with a value of 10, and is decremented on each pass through the loop. Once again there is a time delay so that the countdown takes place at a civilised speed. This is a particularly useful way of slowing the count down, and it uses the instruction PAUSE 20. PAUSE must be followed by a number, and a count of 20 gives a one second delay, with other times proportional so that PAUSE 40 gives two seconds and so on.

Loops and Decisions

It's time to see loops being used rather than just being demonstrated. A simple application is in totalling numbers. The action that we want is that we enter numbers and the computer keeps a running total, adding each number to the total of the numbers so far. In such a program, we would not normally want to total a fixed set of numbers, and it would be more convenient if we could just stop the action by signalling to the computer in some way, perhaps by entering a value like 0 or 999.

A value like this is called a 'terminator', something that is obviously not one of the normal entries that we would use, but just a signal. For a number-totalling program, a terminator of zero is very convenient, because if it gets added to the total it won't make any difference. We detect the terminator by using a conditional test of the type that we have used previously.

Now if all of that sounds rather complicated, take a look at this simple listing ('pro3x5'):

```
pro3x5:
local tot,n,m$(90)
m$="The program will total numbers
for you until you enter 0. Press
any key to start."
view(1,m$)
do
input n
tot=tot+n
until n=0
print"Total:",tot
get
```

The variables are declared, including a long string that contains instructions, and which will be displayed in scrolling form using VIEW. The instructions appear first, and we rely on the LOCAL declaration to make the variable TOT equal to zero before the loop starts. Each time you type a number, then, in response to the request, the number that you have entered is added to the total, and the UNTIL test checks for the entered number N being zero.

Floating point numbers have been used so that you can total any numbers. If N is not zero, the loop is repeated, adding another number to the total, but when N=0 is found (after adding it, which is one of the reasons why 0 is used and not 999) then the loop ends, and the value of TOT is displayed. If you press EXE without having typed a number, then the program takes this as an oversight on your part, and the request for a number to be entered is repeated.

I said earlier that tests for exact equality should not be carried out on floating point nunbers since they aren't necessarily held precisely. However, this program disobeys that rule. The reason why it is acceptable to disobey it in this example is because the number being tested is zero which can be held exactly. Therefore, you can be sure that assigning zero to a variable and then testing the variable to see if it is equal to zero will always give a TRUE result.

Besides being used in UNTIL tests, we saw earlier that conditional tests also follow the IF keyword. Table 3.1 illustrates the type of tests that you can perform using IF.

IF A=B	A exactly equal to B
IF A>B	A greater than B
IF A<B	A less than B
IF A>=B	A greater then B or equal to B
IF A<=B	A less than B or equal to B
IF A<>B	A not equal to B

Table 3.1. The mathematical signs that are used with IF for comparing numbers and number variables.

These use the mathematical signs for convenience, but remember that all of these signs will have a meaning for strings as well, as we have seen. All of these tests will be used in lines that will take the general form:

```
IF (test or tests)
        Action to be done
ENDIF
```

and the action that is placed between IF and ENDIF will be carried out if, and only if, the test is TRUE.

What ELSE?

IF...ENDIF forms a test which can be very useful in programs. There's another extension to IF...ENDIF, however. You can use the word ELSE to carry out a different sort of action. An example makes this a lot clearer, so take a look at this listing ('pro3x6'):

```
pro3x6:
local a$(1),n%
print" Heads or Tails"
pause 40
cls
do
print"Use e to end"
a$=get$
```

```
n%=rnd*2+1
if n%=1
  print"Heads"
  else
  print"Tails"
endif
pause 20
until lower$(a$)="e"
```

This is a simple heads-or-tails gamble, with no scoring. The early lines set things up as usual, then we start at the DO loop of repeated actions that holds the main instructions. You are asked to press a key for the H or T choice (the key you press has no effect on this choice), or use E to end the program. The RND line is the important gambling statement which picks a number that can be one or two (see Chapter Two) which is suitable for a heads or tails choice.

Note that the use of the word INT is not necessary – conversion to an integer will automatically be performed because an integer variable is being assigned to. The test is made following the RND step, so that if N% is one, the word 'HEADS' is printed, and if it's not one, then 'TAILS' is printed. Notice that the test uses LOWER$(A$), which allows you to press 'e' or 'E' with the same effect. In this example, ELSE is being used to choose the alternative action. Normally in an IF test, the alternative is whether the next line(s) is/are run. For example, if you have the following:

```
IF X=4
Y=100
ENDIF
```

the test will be made, and its result will be either TRUE or FALSE. If the result is TRUE (X is four), then the next line is run and program assigns the value of 100 to Y. If the result of the test is FALSE (X is not four), then the next line is ignored, and the assignment is not made. The program moves straight on to the line following ENDIF. Using ELSE allows you to put another option into the test. You can also use the combined word ELSEIF, meaning that the first test has failed and you are now forming another test, as distinct from an alternative action.

```
pro3x7:
local a$(1),n%,sc%,tr%
print" Heads or Tails"
pause 40
cls
do
print"Press h or t"
print"Use e to end"
a$=get$
n%=rnd*2+1
if a$="e"
  stop
elseif n%=1 and a$="h"
  sc%=sc%+1
  elseif n%=2 and a$="t"
  sc%=sc%+1
endif
tr%=tr%+1
print sc%,"out of",tr%
pause 20
until a$="e"
```

The listing above ('pro3x7') shows this in use, with scoring added to the previous example using SC% for the score of correct guesses and TR% for the number of tries. This time, you have to enter h or t to guess heads or tails, or use e to end the game. No LOWER$ test has been used this time, so that you have to be sure that the keyboard is set up to make the letter keys give lower case letters. If you wanted to be sure, you could add to the program before the loop starts the line:

```
KYSTAT 2
```

which forces the keyboard to deliver lower case letters for a letter key. This is how programs such as CALC change the keyboard into its numeric input form (using the equivalent of KYSTAT 4). For more on KYSTAT, see Chapter Six.

The test lines start with the main IF, which tests for the e key being pressed and cause an immediate STOP if this is so. The next line is an ELSEIF which tests for the combination of N%=1 (heads) and A$="h" (your guess of heads). If this is true, then SC% is incremented to bump

up the score. If it's not true, then another ELSEIF causes another test, for N%=2 (tails) and A$="t" (your guess of tails), another combination that calls for the score to be incremented. If neither of these tests is true, then the lines following ENDIF are executed straight away and the score lines are missed out.

The number of tries, TR%, is always incremented whatever the results of the tests, as it lies outside of the IF structure. The procedure then prints out the number of successes and the number of tries for you. This print out is not done if the e key has been pressed, because STOP completely stops the procedure.

Sometimes you find that you need to break out of a DO...UNTIL loop before a count has been completed. You might, for example, have a number of inputs in the course of a loop that allows 1000 inputs, and want to end after only 20. Another common option is to have a title followed by a time delay loop that waits for 25 seconds, but which allows you to break out by pressing any key if you don't want to wait that long. This can be done by a test that includes logic AND or OR, using the form of test that was illustrated above. In either example, you can form a test like:

```
UNTIL K%=1000 OR KEY<>0
```

so that the loop will be terminated either by the count or by pressing any key. If you control timing by means of PAUSE, then you can also modify this time delay to break out by pressing any key. The method is to use a negative number after the PAUSE statement, instead of a positive one, so that PAUSE -100 would wait for five seconds *or* until any key was pressed, as compared to PAUSE 100 would force you to wait for the full five seconds.

One point of warning here is that when you use PAUSE in this way, the code for the key that was pressed remains stored and can affect any GET line that you might have later. When you use PAUSE with a negative number, always make the following instruction KEY, which will dispose of the stored code without any effect on the program.

There is one instruction that can cause a loop to end at any point, not just the start or the end. The instruction BREAK will always end a loop, and it can be put as part of an IF test, anywhere in the middle of the loop. BREAK can be used in the DO...UNTIL type of loop, or in the WHILE...ENDWH type (details follow), and its use is illustrated in later examples.

WHILE and ENDWH

The DO...UNTIL type of loop makes its test at the end of the loop when the UNTIL instruction is reached. This means that everything inside the loop will be executed at least once, because you can't get to the end of the loop until you have carried out all the instructions between DO and UNTIL. OPL offers you, as an alternative, a very different type of loop, the WHILE...ENDWH. This can sometimes make it much easier to program a loop, and is essential if you want to make sure that the loop instructions do not run at all if the test gives a FALSE result. The principle is that you start your loop with a condition, then you have as many lines as you like of what has to be done in the loop, and finally, the word ENDWH (meaning END WHile) to mark the end of the loop.

Yes, an example would certainly help, so cast an eye on this listing ('pro3x8'):

```
pro3x8:
local t,j
j=1
while j<>0
print"Number",
input j
t=t+j
print"Total:",t
endwh
print"END"
pause -40
```

This is another version of an old friend, the number-totalling program. This time we have J=1 near the start. This is needed because of the way that a WHILE...ENDWH loop works, as we'll see. The start of the loop is at WHILE J<>0. What this means is that the loop will be repeated for as

long as J is not zero. When the program starts, however, you will not have input any number J by this stage. This is why a 'dummy' value for J has to be assigned before the loop starts. Without this J=1 step, the program would finish as soon as it got to the WHILE test. This is something that you have to be careful about, particularly if you have used any of the older versions of BASIC that did not have the WHILE...ENDWH (or WHILE...WEND) loop. If you find, when you run a program, that a WHILE...ENDWH loop appears not to run, then this is the first thing to suspect.

The steps in the loop are familiar, and we needn't go over them again. The important one to note is the ENDWH line. This marks the end of the loop, and will automatically send the loop back to the WHILE test. There's no need for IF tests, and you can even nest WHILE...ENDWH loops inside each other. The only snag is in remembering that the test in a WHILE...ENDWH loop is made right at the start of the loop. You must have a value for whatever is being tested at this stage, or the loop simply won't run at all.

```
pro3x9:
local n%
print"Please enter number"
print"-range 1 to 5 only"
input n%
while n%<1 or n%>5
  print"Out of range"
  print"1 to 5 only"
  input n%
endwh
print"You picked",n%
get
```

The above listing ('pro3x9') shows yet another use for the WHILE...ENDWH loop. In this case, it acts as a 'mugtrap'. A mugtrap (polite name – data validator) is a piece of program that tests whatever you have entered. If what you have entered is unacceptable, like a number in the wrong range, then the mugtrap refuses to accept the entry, shows by a message on the screen why the entry is unacceptable, and gives you another chance to enter something better.

Mugtraps are very important in programs where a piece of incorrect entry might stop a program with an error message.

In this example, then, you are invited to enter numbers in the range one to five. If the number that you enter is in this range, all is well, but if not (try it!), then the WHILE...ENDWH loop swings into action. This prints an error message of your own, and gives you another chance to get it right.

That's the essence of a good mugtrap, and the WHILE...ENDWH loop is ideal for forming such traps. Note, despite the emphasis on numbers in some examples, that the WHILE...ENDWH loop is just as much at home with strings. You can have lines like:

```
WHILE Name$ <>"X"
```

to allow you to keep entering names into a list, or:

```
WHILE UPPER$(AN$)<>"Y" AND UPPER$(AN$)<>"N"
```

to make a mugtrap for a 'Y' or 'N' answer. Don't forget the use of UPPER$ or LOWER$ to avoid having to test for y and n as well as for Y and N, and BREAK to end a loop in midstream.

Last Pass

When you start writing programs for yourself, designing a loop often appears to be difficult. It's not, but you need to approach it with some method. The best way is to write down what the loop conditions are to start with. What conditions do you want at the start of the loop, for example? If the loop must run at least once, then a DO...UNTIL should be used. If the loop must not run unless conditions are correct at the start, then use the WHILE...ENDWH loop.

Once again, though, you have to look for the starting conditions. Remember that the WHILE...ENDWH loop makes a test right at the start of the loop, and so you have to ensure that whatever is tested at the start has a suitable value, this is not the case with the DO...UNTIL type. If you might need to make a test that would end a loop mid-way, you can use the IF test along with BREAK.

Next, you have to think of what is to be done in each pass of the loop. This might be some string action, or some number action, or a bit of both. The really important bit, however, is to decide what will end the loop. You might want to end a counting loop before the true end of the count, in which case please heed the words of wisdom earlier in this Chapter.

The really important thing here is that if you are using a count in the loop, you must remember to increment (or decrement) the counting variable, otherwise the loop can never end, and you will be faced with a lot of stabbing at the ON/CLEAR and Q keys.

Finally, it's possible to get yourself tied up in testing a loop. Suppose, for example, you have designed a loop that calls for the entry of 5000 names, and you want to check that it ends correctly when you enter an "X", or after 5000 entries. Needless to say, you don't check it by going through all 5000 entries, a much easier way is to alter the count number to something smaller, like five, for testing.

String Functions

If numbers turn you on, then what we have done so far in this Chapter will be of use, but string functions are in many ways more interesting. What makes them that way is that the really eyecatching and fascinating actions that the computer can carry out are so often done using string functions. What's a string function, then? As far as we are concerned, a string function is any action that we can carry out with strings. That definition doesn't exactly help you, I know, so let's go into more detail.

A string, as far as OPL is concerned, is a collection of characters which is represented by a string variable, that is a name which ends with the dollar sign. You can pack practically as many characters as you are likely to need into a OPL string – a maximum of 255 characters per string is allowed, which is a longer string than most of us will ever want to use.

Each string variable that you use, however, must have its maximum number of characters declared in advance, and for a 20-character screen width, it doesn't make sense to have long strings for anything you are going to print unless you are using the VIEW type of display instruction.

You can assign characters to a string variable by using the equality sign. When you assign in this way, you need to use quotes around the characters. You can also assign using INPUT, along with a string variable name, when no quotes are needed.

Like other computers, OPL stores its strings in a way that is very different from the way that is used to store numbers, making use of what is called ASCII code. The letters stand for American Standard Code for Information Interchange, and the ASCII (pronounced Askey) code is one that is used by most computers.

Figure 3.1 on the following page shows a printout of the ASCII code numbers and the characters that they produce on a Star LC24-10 dot-matrix printer, with the printer in its U.S. character set.

Each character is represented by a number, and the range of numbers is from 32 (the space) to 127. On the printer, 127 produces nothing, but on the Organiser screen, you'll see a small left-pointing arrow appear.

Now all number variables are represented in a different type of coding, one that uses the same number of byte codes no matter whether the value of the variable is large or small. There's one type of number coding for integers, and another for ordinary floats. Because a string consists of a set of number codes in the memory of the computer, one code for each character, we can do things with strings that we cannot do with numbers. We can, for example, easily find how many characters are in a string. We can select some characters from a string, or we can change them or insert others. Actions such as these are the actions that we call 'string functions'.

No.	Char	No.	Char
32	(space)	80	P
33	!	81	Q
34	"	82	R
35	#	83	S
36	$	84	T
37	%	85	U
38	&	86	V
39	'	87	W
40	(	88	X
41	)	89	Y
42	*	90	Z
43	+	91	[
44	,	92	\
45	-	93	]
46	.	94	^
47	/	95	_
48	0	96	`
49	1	97	a
50	2	98	b
51	3	99	c
52	4	100	d
53	5	101	e
54	6	102	f
55	7	103	g
56	8	104	h
57	9	105	i
58	:	106	j
59	;	107	k
60	<	108	l
61	=	109	m
62	>	110	n
63	?	111	o
64	@	112	p
65	A	113	q
66	B	114	r
67	C	115	s
68	D	116	t
69	E	117	u
70	F	118	v
71	G	119	w
72	H	120	x
73	I	121	y
74	J	122	z
75	K	123	{
76	L	124	|
77	M	125	}
78	N	126	~
79	O	127	
80	P		

Figure 3.1. The ASCII character set for the range 32 to 127, as printed by a STAR LC24/10. This printer can also print characters for the ranges 1 to 31 and 128 to 255.

LEN in Action

One of these string function operations that I mentioned was finding out how many characters are contained in a string. Since a string can contain up to 255 characters, an automatic method of counting them is rather useful, and LEN is that method. LEN has to be followed by the name of the string variable, within brackets, and the result of using LEN is always an integer number so that we can print it or assign it to an integer variable. You don't need to put a space between the 'N' of LEN and the opening bracket.

```
pro3x10:
local n%,s$(30)
s$="OPL in action"
n%=(21-len(s$))/2
at n%,2
print s$
get
```

The listing above ('pro3x10') shows a simple example of LEN in use. This program uses LEN as part of a routine which will print a string called S$ centred on a line. This is an extremely useful routine to use in your own programs, because its use can save you a lot of tedious counting when you write your programs. The principle is to use LEN to find out how many characters are present in the string S$. This number is subtracted from 21, and the result is then divided by two. If the number of characters in the string is an even number, then the .5, produced by the division by two, will be ignored because N% is an integer variable.

You can, incidentally, use 21 or 22 as the characters per line. Whichever one you use, you will find that words are reasonably well centred – 22 works better with phrases which have an even number of characters, and 21 works better with phrases which have an odd number of characters. Yes, you could program for that sort of variation, but one thing at a time, please!

We can use a routine of this type to centre anything that has the name S$. In Chapter Four, we'll be looking at the idea of using procedures

more fully, allowing you to type the set of instructions (for centring a title, for example) just once, and then use them for any string that you like.

NUM$ and VAL

You know by now that there are some operations that you can carry out on numbers but not on strings, and some which you can carry out on strings but not on numbers. This might be inconvenient, but as it happens, we can convert from one form into another quite easily. This allows us to perform arithmetic on a number that has been in string form, and also to use string functions on a number that was formerly only in number form. Take a look at this listing ('pro3x11'):

```
pro3x11:
local v%,n$(6),v$(2)
n$="22.5"
v%=2
print n$,"*",v%,"is",v%*val(n$)
pause 40
v$=num$(v%,1)
print len(v$),"character(s)"
pause 40
print"but adding gives",
print n$+v$
get
```

To start with, we declare two strings and a number variable, and then make N$ a number in string form, and V% a number in integer number form. The program then shows how we can carry out arithmetic with N$. By typing VAL(N$) in place of N$ alone, the number value of N$ is used in the calculation, and the correct result is obtained. Next, number V% is transformed into a string, V$, by the use of NUM$(V%). As the line using LEN shows, a single-digit number becomes a one-character string.

The last line is there to remind you of what can happen if you forget about VAL and try to add two strings! Note that NUM$ requires you to state how many characters are needed for the number as well as supplying the variable that holds the number. In addition, it always delivers the string form of an integer, even if a floating point number has been converted.

NUM$ also 'fields' a number. This means that it prints a number in a given space, set either to the left or the right of the space, according to the value of the second number within the brackets. Three other functions, FIX$, GEN$ and SCI$ also carry out this form of conversion, as the following listing ('pro3x12') illustrates.

```
pro3x12:
local v
v=714.639
print "fix$"
print fix$(v,1,8)
print fix$(v,1,-8)
pause 40
print "gen$"
print gen$(v,6)
print gen$(v,-6)
pause 40
print "sci$"
print sci$(v,2,9)
print sci$(v,2,-6)
get
```

FIX$ allows you to state the number of decimal places and the total length of the string, using the form:

FIX$(number, decimal places, total length).

If the total length number is negative, then the result will be placed to the right of the 'field', the space that is allocated for it. GEN$ will fit a number into the specified field space, using the most appropriate representation so that as precise a form of the number as possible will be used. The result can therefore appear as integer, float or scientific form according to what will fit best. When SCI$ is used, the result is always in scientific notation and as the example shows for SCI$, if the field space is inadequate, only a string of asterisks will appear.

VAL is used mainly when you have had an INPUT to a string variable, and after testing for items like the length of the string. The string

variable is then converted to number form using VAL so as to be tested for correct range. If necessary, a number that will eventually be an integer can be converted into float form for its range test. This will ensure that the program is not stopped by an error message if the size of the number is outside the range of an integer. If testing shows that the number is acceptable, it can be changed into integer form by a statement such as X%=X. The use of NUM$ and its associates is less common, but can be applied when numbers have to be placed in strings with a suitable format.

A Slice in Time

The next group of string operations that we're going to look at are called slicing operations. The result of slicing a string is another string, a piece copied from the longer string. String slicing is a way of finding what letters or other characters are present at different places in a string. You can also use it to select different parts of a string so that these can be printed or reassigned.

```
pro3x13:
local a$(8),b$(4),c$(6),d$(11)
a$="Oriental"
b$="Post"
c$="Likely"
D$="programming"
print left$(a$,1)+left$(b$,1)+left$(c$,1),
print left$(d$,7)
get
```

All of that might not sound terribly interesting, so take a look at the listing above ('pro3x13'). A set of strings are assigned in the first four lines, using variable names A$ to D$. What's printed at the end is a phrase which uses letters that have been selected from the left-hand sides of some of the strings.

Now how did this happen? The instruction LEFT$ means 'copy part of a string starting at the left-hand side'. LEFT$ has to be followed by two quantities, within brackets and separated by a comma. The first of these is the variable name for the string that we want to slice, A$ in the first of the lines. The second is the number of characters that you want to slice (copy, in fact) from the left-hand side. The effect of LEFT$(A$,1) is therefore to copy the first character from "Oriental", giving 'O'. The next string slice is obtained by using LEFT$(B$,1) which takes 'P' from 'Post', and so on.

The first three string slices are tacked together (concatenated), and the last slice is printed separately, using a comma to make a space. The result of all this is the phrase which is printed on the screen.

String slicing isn't confined to copying a selected piece of the left-hand side of a string. We can also take a copy of characters from the right-hand side of a string. This particular facility isn't used quite so much as the LEFT$ one, but it's useful none the less. The listing below ('pro3x14') illustrates the use of this instruction to avoid having to type a word over again.

```
pro3x14:
local a$(9)
a$="OPL magic"
print"Its all",
print right$(a$,5)
get
```

There are, of course, more serious uses that this. You can, for example, extract the last four figures from a string of numbers like 010-242-7016. I said a string of numbers deliberately, because something like this has to be stored as a string variable rather than as a number. If you try to assign this to a number variable, you'll get a silly answer. Why? Because when you type N = 010-242-7016 then the computer assumes that you want to subtract 242 from 10 and 7016 from that result. The value for N is -7248, which is not exactly what you had in mind! If you use N$="010-242-7016" then all is well.

There's another string slicing instruction which is capable of much more than either LEFT$ or RIGHT$. The instruction word is MID$, and it has to be followed by three items, within brackets, and using commas to separate the items. The first item is the name of the string that you want to slice, as you might expect by now. The second item is a

number which specifies where you start slicing. This number is the number of the characters counted from the left-hand side of the string, and counting the first character as one. The third item is another number, the number of characters that you want to slice, going from left to right and starting at the position that was specified by the first number.

```
pro3x15:
local a$(9),b$(11),c$(8)
a$="Organiser"
b$="Programming"
c$="Language"
print mid$(a$,4,2)+mid$(c$,6,2)+mid$(b$,5,3)
get
```

The listing ('pro3x15') above demonstrates shows how this can pick out groups of letters which are not from the left or the right of existing strings. The syntax of MID$ is:

```
MID$(string to slice, starting place, no of chars to copy)
```

so that the brackets contain one string variable and two integers or integer variables.

Inside, Upstairs and Downstairs

There are some string functions that are not so easy to place in groups, but one, LOC, definitely belongs close to the slicing functions. The use of LOC is to find if one string is contained within another. The syntax is:

```
LOC(main string, short string)
```

and the result is a number value.

The number is the position number in the main string where the short string starts. If the short string does not exist in the main one, then LOC gives zero, and this number-or-zero choice can be used in a very simple way to pick out strings, as the next listing ('pro3x16') illustrates.

```
pro3x16:
local a$(30),b$(30),c$(30),f$(30)
a$="Effective Management"
b$="How to Manage Purchasing"
c$="Keeping a Menagerie"
f$="Manage"
print loc(a$,f$)
print loc(b$,f$)
print loc(c$,f$)
get
```

In this example, three strings contain phrases which might be book titles and the program is set to work to scan the titles looking for a word 'Manage'. The test is arranged so that if the word 'Manage' is not found, the result will be zero – in a later Chapter we shall see how this could be used to print different reports on the items. Note incidentally, that if the strings to be searched contain 'Manage' and you are looking for 'manage', then LOC will still find the word – the LOC action is not case-sensitive.

Talking of upper and lower case letters, OPL has the functions UPPER$ and LOWER$ to convert all the characters of a string to one form or the other. These actions refer specifically to letters, not characters generally, and only the letters of the alphabet are affected. Using UPPER$ is useful in comparing strings, because if you are searching for Smith, it's useful to know that the entries of SMITH, smith and even sMith will be correctly matched!

The provision of UPPER$ is therefore very useful for writing programs that search for matching strings, and also for arranging strings into correct alphabetical order.

Note that the case of the characters in the original string is not changed – the use of UPPER$ or LOWER$ need only affect comparisons, though you can write lines such as:

```
K$=UPPER$(K$)
```

to convert all the characters to upper case and retain the same string name.

The REPT$ action is used when you want a character or set of characters to be used to fill a string. Suppose, for example, you want to print a line of 20 asterisk marks. You could type all of these 20 asterisks in a PRINT line, but OPL allows you to do this by using:

```
PRINT REPT$("*",20)
```

which is a lot more compact. You can use a string variable for the first item in the brackets a any number variable for the second, assuming that they have been correctly assigned. You can also assign the resulting string, using the form:

```
X$ = REPT$(A$,N%)
```

if the string variables have been declared correctly.

More Priceless Characters

It's time now to look at some other types of string functions. If you hark back a few pages, you'll remember that we introduced the idea of the ASCII code. This is the number code that is used to represent each of the characters that we can print on the screen. We can find out the code for any letter by using the function ASC, which is followed, within brackets, by a string character in quotes or a string variable (no quotes). The result of ASC is a number, the ASCII code number for that character. If you use ASC("OPL"), then you'll get the code for the 'O' only, because the action of ASC includes rejecting more than one character. The following listing ('pro3x17') shows this in action.

```
pro3x17:
local a$(1)
input a$
print asc(a$)
get
```

You can enter any character from the keyboard and then see its ASCII code printed on the screen. Another way of doing this is to switch to Calc, and enter the character following a % sign, so that entering %p would give 112, the ASCII code for p.

ASC has an inverse function, CHR$. What follows CHR$, within brackets, has to be a code number, and the result is the character whose code number is given. The instruction PRINT CHR$(65), for example, will cause the letter A to appear on the screen, because 65 is the ASCII code for the letter A. We can use this for coding messages. Every now and again, it's useful to be able to hide a message in a program so that it's not obvious to anyone who reads the listing.

Using ASCII codes is not a particularly good way of hiding a message from a skilled programmer, but for non-skilled users it's good enough. There is an example of this in Chapter Five.

The CHR$ action, however, can do rather more than this suggests. There are several ASCII codes that do not correspond to characters that can be printed on the screen, and CHR$ allows us to use them. One example is the use of CHR$(34) to put a quotemark into a PRINT line, something we can't do directly because the quotemark usually indicates the end of a string to be printed. Using PRINT CHR$(16) will sound the buzzer – try it.

Another 'invisible' character is CHR$(27). This is called the ESC character, and is the same as is generated by the key marked ESC on some desktop machines. By LPRINTing strings that contain CHR$(27), you can carry out many effects that make control of your printer possible, and by PRINTing them you can send control codes to the Organiser itself. The manual for OPL has only a brief mention of these codes but they can be very useful in obtaining special effects such as positioning the cursor.

The Law about Order

We saw earlier that the symbols '=', '<', '>' can be used to compare numbers. We can also compare strings, using the ASCII codes as the basis for the comparison. Two letters are identical if they have identical ASCII codes, so it's not difficult to see what the identity sign, = , means when we apply it to strings.

If two long strings are identical, then they must contain the same letters in the same order. It's not so easy to see how we use the > and < signs until we think of ASCII codes. The ASCII code for A is 65, and the code for B is 66. In this sense, A is 'less than' B, because it has a smaller ASCII code. If we want to place letters into alphabetical order, then, we simply arrange them in order of ascending ASCII codes. This is not totally straightforward, because the ASCII codes for the lower case letters are greater than the ASCII codes for the upper case letters. This would lead to the letter Z being placed before the letter a if we did not use UPPER$ on each string we compared. Languages with no UPPER$ statement are at a considerable disadvantage here!

This process can be taken one stage further, though, to comparing complete words, character by character. The next listing ('pro3x18') illustrates this use of comparison using the '=' and '>' symbols, and making use of tests that will be explained in more detail in the following Chapter.

```
pro3x18:
local a$(20),b$(20),c$(20)
a$="qwerty"
print"Type a word",
input b$
if a$=lower$(b$)
print"Same as mine"+chr$(33)
pause 20
stop
endif
if a$>lower$(b$)
c$=a$
a$=b$
b$=c$
endif
print"Order is - "
print a$
print b$
get
```

The first line assigns a nonsense word – it's just the first six letters on the first row of a typewriter. You are then asked to type a word. The comparisons are then carried out and if the word that you have typed, which is assigned to B$, is identical to qwerty, ignoring differences in case, then the message about the words being the same is printed, and the program ends. If qwerty would come later in an index than your word, then the contents of the variables are swapped. The variable C$ is used to hold the data from A$, then A$ is assigned with the contents of B$. Finally, B$ is assigned with the contents of C$ which came originally from A$.

If, for example, you typed peripheral, then since Q comes after P in the alphabet and has an ASCII code that is greater than the code for P, your word B$ scores lower than A$, and these three lines swap them round. The last line will then print the words in the order A$ and then B$, which will be the correct alphabetical order.

If the word that you typed comes later than qwerty, for example, tape, then A$ is not 'greater than' B$, and the test for swapping fails. No swap is made, and the order A$, then B$, is still correct.

Note the important point though, that words like qwertz and qwetx will be put correctly into order – it's not just the first letter that counts. By using LOWER$ in this example, the order will not be changed by using capital letters; we could just as easily and effectively have typed QWERTY in capitals and used UPPER$ for the conversion. The LOWER$ conversion affects only the swap, not the display of words.

Complex Data – Put it on the List

The variable names that we have used so far are useful, but there's a limit to their usefulness. Suppose, for example, that you had a program that allowed you to type in a large set of numbers. It would be very unsatisfactory if you had to assign a new variable name to each item, and if you had to put in an INPUT line for each and every input.

It would be much more satisfactory, in fact, if you could have just one INPUT routine that could be used in a loop, but to do this we need a

different type of variable. The following listing ('pro3x19') illustrates this, and we'll ignore the first line for the moment.

```
pro3x19:
local a%(10),n%
n%=1
while n%<11
  a%(n%)=100*rnd+1
  n%=n%+1
endwh
print"  Marks List"
print
n%=1
do
print"Item",n%,
print"gets",a%(n%),
print"marks"
n%=n%+1
until n%>10
get
```

The first loop generates an (imaginary) set of examination marks. This is done simply to avoid the hard work of entering the real thing, and in this example RND*100+1 is used to generate a number whose value will lie somewhere between 1 and 100 – there is no need to use INT here because the number is being assigned to an integer variable. The variable that is used to hold the mark number is something new, though. It's called a 'subscripted variable' or 'array element', and the 'subscript' is the number that is represented by N%. The name 'subscripted' that we use has nothing to do with computing, it's a name that was used long before computers were around.

How often do you make a list with the items numbered 1,2,3.. and so on? These numbers 1,2,3 are a form of subscript number, put there simply so that you can identify different items. Similarly, by using variable names A(1), A(2), A(3) and so on, we can identify different items that have the common variable name of A. A member of this group like A(2) has its name pronounced as 'A-of-two'.

The usefulness of this method is that it allows us to use one single variable name for the complete group, picking out items simply by their identity numbers. Since the number can be a number variable or an expression, this allows us to work with any item of the group. The example shows the group being constructed in a loop with each item being obtained by finding a random number between one and 100, and then assigned to A%(N%). Since A% is an integer name, this is an integer array, and the N% is the subscript number which must be an integer.

You can have arrays of any data type, number or string, but the subscript numbers must always be integers. If you use any other type of number variable as the subscript, it will be chopped to an integer, so don't expect to be able to refer to item A(2.5)!

In the example, ten of these 'marks' are assigned in this way, and then the second loop prints each of them out. It makes for much neater programming than you would have to use if you needed a separate variable name for each number.

The LOCAL line has prepared the computer for the use of subscript numbers from 1 to 10, but no higher. You must not attempt to use A%(0), nor can you use A%(11) or any higher number. You will get an error message if you do so, but only when the program runs, not when it is translated.

The important part of the LOCAL instruction, then, consists of naming each variable that you will use for arrays, and following the name with the maximum number, within brackets, that you expect to use. You aren't forced to use this number, but you must not exceed it. If you do, and your program stops with an error message, you will have to change the LOCAL instruction and start again – which will be tough luck if you were typing in a list of 100 names!

```
pro3x20:
local a(10),n%
n%=1
while n%<11
  a(n%)=int(rnd*100)+1
  n%=n%+1
endwh
n%=n%-1
```

```
print"There are",n%,"items"
print"Total is",sum(a(),n%)
print"Mean is",mean(a(),n%)
pause -50
print"Largest is",max(a(),n%)
print"Smallest is",min(a(),n%)
print"S.D. is",std(a(),n%)
pause -50
```

The listing given above takes this use of array variables another step further. This example again assigns a random number between one and 100 to each of ten array elements. When this task is complete, some functions are used, not on a single variable but on the whole array.

There is a complete set of these 'list' functions, all of which are also used in the Calculator, and they can act on a set of items of any type given explicitly in a list, or on items in an array. There is one restriction, however. If you use items in an array, the array must be of floating point type. In other words, if your array is N%(A%), you have to use other methods, but if it's N(A%) then all is well.

The syntax for any list function can be illustrated by using SUM. The list form is X=SUM(A,B%,6,5.5) – with a mixture of integer and floating point numbers and variables in the list within the brackets. The alternative syntax is X=SUM(A(),10), which will take the sum of the first ten items in the array called A, a floating point array. If you try to use an integer array you will get the SYNTAX ERR message.

In this example, a set of numbers is generated at random, and these functions are used to find the total, the average, the largest item, the smallest item and the standard deviation (which is the average difference from the mean for all the items). The printed lines are arranged in sets of three, with a PAUSE to hold each set in place long enough to read it – you could use a GET in these places to give yourself more time, or use a PAUSE with a larger (negative) number.

Remember that using PAUSE with a negative number allows you to end the pause by pressing a key.

Table 3.2 provides a summary of the list functions and their actions.

Function	Application
MAX(x,y,z...)	Finds maximum in list
MEAN(x,y,z..)	Finds average of list of numbers
MIN(x,y,z...)	Finds minimum in list
STD(x,y,z..)	Finds standard deviation of list of numbers
SUM(x,y,z..)	Finds sum total of list of numbers
VAR(x,y,z..)	Finds variance of list of numbers

Table 3.2.The list functions of OPL.

Note: The list is shown as (x,y,z..) which can use any mixture of constant numbers (like 2.6), integer variables (like k%) and float variables. The list can also consist of a floating-point array, with nothing placed between the brackets, but the number of items stated following a comma, like SUM(A(),20)

```
pro3x21:
local n$(12,20),n%,j%
n%=1
while n%<11
  print"Surname",n%
  input n$(n%)
  n%=n%+1
  cls
endwh
n%=1
while n%<11
  j%=1
  cls
  do
  print n$(n%)
  n%=n%+1
  j%=j%+1
  until j%>4
  get
endwh
pause -50
```

The use of arrays is not confined to numbers, and the listing above ('pro3x21') illustrates a string array. This has to be prepared for in the

LOCAL line with a pair of numbers, one for the number of strings and a second one for the maximum number of characters in a string. A WHILE loop is then used to enter surnames of not more than 20 characters each, and the dimensioning provides for up to 12 of these, though only 10 will be used.

After the names have been entered they are printed in batches of four, with a 'press-any-key' stage to allow time to read each set. This is done by using an inner loop with its own counter J% which prints in sets of four and then returns to the main loop to reset J%. The main counter N% is incremented each time a name is printed and the loop allows for only 10 such items being printed. We needed to allow for 12 in the array, however, because the inner loop operates in sets of four, and the third set of four will make N% increment to 12 even though the outer loop has specified an end after 10 items – this is because the WHILE test cannot be made while the inner loop is executing.

Manipulating Arrays

The main point of using arrays is to manipulate data that is held in the form of array items. For number arrays, we have seen how the list functions of OPL allow for most of the useful manipulations of number arrays to be carried out. Manipulation of string arrays can perform almost anything that you want to do with the data, but it usually boils down to two particular actions, searching and sorting.

More has been written on these actions than on anything else in computing, but a lot that has been written was written a long time ago, and some of it is very hard to follow unless you happen to be a student of computing theory. In this Chapter, we can't exactly go into great detail about these actions, but we'll look at one example each of methods that we can use. These methods apply equally to any type of array, but you have to make suitable adjustments.

For example, if you make a test for a number array that uses A%, then adapting the program to a string array that uses A$ means that the wording of the test must be changed. For that reason, I'll demonstrate one number action and one string action.

We'll start with searching. There's no problem in searching an array for the 56th item – you just do a PRINT A%(56), or whatever you want. Even if this means an input step, it amounts only to:

```
INPUT X%
PRINT A%(X%)
```

and is no problem. The searching arises when you want to know if you have any items that are divisible by seven, or any names that start with 'R'. This involves looking at each item in the array and testing it, which is what searching is all about. The simplest type of search involves looking through all of the items, but if the items are arranged in some order it's sometimes possible to devise quicker and more elaborate searches, called binary searches.

Leaving these complications for the more advanced programmer, let's see what a search through a number list looking for numbers divisible by seven would be like. The example that follows ('pro3x22') first generates an array using random numbers, and you can see by the time it takes to produce the 'List ready' report that this takes a fairly short time.

```
pro3x22:
local a%(100),n%
n%=1
do
  a%(n%)=rnd*100+1
  n%=n%+1
until n%>100
print"list ready"
n%=1
do
  if a%(n%)/7.0=a%(n%)/7
  print a%(n%);
  print "   ";
  endif
  n%=n%+1
until n%>100
get
```

Since the list consists of numbers generated almost at random, they are in random order and some of them will probably be divisible by seven. The search uses a loop which tests each item on the list by using the line:

```
IF A%(N%)/7.0=A%(N%)/7
```

to detect divisibility by seven. By using in the first half the dividing number in the form 7.0 we force the division to give a floating point answer, so that for a number such as 50, the result will be 7.1428571. This is compared to the integer result, obtained by using seven rather than 7.0, which for the number 50 would be seven. The two are identical only when the number is exactly divisible by seven – this is a very useful way of checking for multiples.

If the test is true, and the number does divide evenly by seven, then the number is printed. Note that the case of A%(N%)=0 would have to be excluded if there were any chance of zeros occurring in the data, as the type of test used here would consider that zero was divisible by seven.

A common mistake in your first effort at searching is to find only the first matching answer, and to stop the search at that point. Unless you know that there can be only one item that answers the description this can be a cause of problems.

If this were a string list, you might be looking for all of the items that started with a particular letter. You would therefore have an input line to find what letter you wanted to look for, and you would assign this to some variable like S$. You would then use a similar loop, but with a test such as:

```
IF LEFT$(S$,1)=LEFT$(J$(N%),1)
```

and following this a line that printed the string items that were found.

Sorting into order is a very much more difficult business. At one time, all books on BASIC would show a routine called the Bubble-sort, whose only merit was that it was easy to explain. Since a Bubble-sort on a long string array can take hours to complete, it's not one that is of much use to programmers, unless you are working with a string array that is almost completely sorted to begin with. The problem is that any sorting routine that is efficient is very difficult to explain, and the best way of understanding it is to go through it step by step.

The faster routine that is the favourite in terms of speed and comparative simplicity is called the Shell-Metzner sort, after the names of its original programmers. It is based on the idea of comparing items that are some way apart in the list, starting by dividing the list into two equal parts, and looking at the first item in each part. These items are compared, and if they are in the wrong order, they are swapped. The spacing is then decreased and the exercise repeated until the items that are being compared are next to each other. Another set is then taken, and the process repeated until all the items of the whole list are in order.

The Organiser has its own built-in sort routine which can be used, for example, to put Notepad items into alphabetical order. This, however, is not made available as a routine in OPL so that if you need to sort a set of items in OPL you have to use your own routine. The listing below ('pro3x23') shows this in action, with a list that is made up from 'words' created at random – and that's a routine that you might want to use for your own tests!

```
pro3x23:
local
w$(100,20),n%,mx%,a$(20),j%,y%,it%,t%,fg%,z%,msg$(80)
mx%=100
msg$="Please wait...creating strings. Press any key to
start."
view(1,msg$)
n%=1
do
  a$=""
  j%=1
  t%=2+rnd*10
   do
   a$=a$+chr$(65+rnd*26)
   j%=j%+1
   until j%>=t%
```

```
  w$(n%)=a$
  n%=n%+1
until n%>100
msg$="List ready - press any key to sort"
view(1,msg$)
cls
y%=1
while y%<mx%
  y%=2*y%
endwh
while y%<>0
  y%=(y%-1)/2
  it%=mx%-y%
  t%=1
   do
   j%=t%
     do
     fg%=0
     z%=j%+y%
     if w$(z%)<=w$(j%)
     a$=w$(z%)
     w$(z%)=w$(j%)
     w$(j%)=a$
     j%=j%-y%
     if j%>0 and y%>0
     fg%=1
     endif
     endif
     until fg%=0
   t%=t%+1
   until t%>it%
endwh
rem End of sort
n%=1
do
  if n%/4.0=n%/4
  get
  cls
  endif
  print w$(n%)
  n%=n%+1
until n%>100
get
```

The sort routine then fixes a number that is a power of two to decide where to start splitting the list, and it then starts making comparisons. If you want to see what items are being compared, then add a line:

```
PRINT Z%,J%
```

along with a pause, but don't keep this in because it greatly slows down the rate of sorting. You can use this routine for sorting numbers rather than strings, simply by substituting a number array for W$ – you could, for example, use NUM%, or NUM for integers or for floats respectively.

4 : Menus and Procedures

Very often we want to present a user with a menu on the screen. A menu is a list of choices, usually of program actions. By picking one of these choices, we can cause a section of the program to be run. The use of menus is catered for in various ways in other versions of BASIC, but OPL introduces its own methods that are particularly suited to the Organiser with its 20-character four-line screen. Using the OPL method makes your own menus look exactly like the types of menu that the Organiser uses in its other actions.

The programming of any menu consists of two parts, the display of the items from which the choice is made, and the way in which the different procedures for these choices are run.

Since OPL provides its own way of displaying items, we can concentrate for the moment on how procedures are used, a point which is central to the way that more advanced OPL programming can be carried out.

This listing ('pro4x1') illustrates this in action:

```
pro4x1:
local t$(20),und$(20),a$(20)
t$="OPL Computing"
centre:(t$,1)
und$=rept$("-",len(t$))
centre:(und$,2)
print
print"Neat"+chr$(33)
get

centre:(a$,n%)
at (21-len(a$))/2,n%
print a$
```

? See 107

When the program runs, three string variable names are declared and a phrase is assigned to the string variable T$. The next line is centre:(T$,1), which means that the program must jump to the routine which starts at the label word centre, and carry the value that has been assigned to T$ with it along with the number value of one. When you see a name ending with a single colon, then you have to look for a matching procedure which carries the same name. This procedure must be separately translated and stored in the RAM, something that sharply distinguishes OPL from any other common form of BASIC. Any procedure in OPL will start with a heading which is its own name, and any other procedure names refer to other procedures which are separately recorded.

The name centre is followed by the usual colon, and then also by a string name and a number within brackets. The string name that is used is the name of an assigned string that the procedure centre will work on. The number also is supplied so that it can be used by centre. These items, in this example a string name and a number, are called 'parameters', and by putting these parameters within brackets when the name of a procedure is used, the parameters are 'passed' to the procedure.

Looking at the separate listing, you can see the procedure centre(A$,N%), which is the procedure that is called up by using the word centre in the main program. This procedure is started as follows:

1) Select New from the Prog menu.

2) Type centre, press EXE.

3) The word centre: now appears with the cursor following it. Type the portion within brackets.

4) Press EXE again to start the typing of the actions of the procedure.

The name is the important thing here, and the quantities within the brackets can be anything you like provided that they are of the same type as the quantities in the brackets used in the centre line of the main (or calling) program. Our calling line used a string and an integer, so the procedure line must also use a string quantity and an integer in the

same order, but they can be given any names that you like to use for a string and an integer variable. In the procedure, the number of characters in A$ is found. The action of calling a procedure includes having values automatically assigned to its parameters. Therefore, when centre is first called, the variable A$ is assigned the value of T$ and N% is assigned the value one. It's just as if you had programmed the lines:

```
A$=T$
N%=1
```

to pass the value that has been assigned to T$ into the variable A$ and one into the variable N%.

The following lines then locate the cursor at a position which will print the value of A$, which is identical to T$, centred on the screen, using line N%. The string is printed and at that point the procedure ends. When this happens, control is returned to the line that called the procedure – most varieties of BASIC require a command RETURN for this action but, as we shall see, OPL uses this word rather differently, more like the way it is used in languages other than BASIC. In this first case, when the centre procedure ends, the return is to the fourth line which carries out an assignment to UND$ of a string of underlining dashes – note how REPT$ has been used here. This time calling centre(UND$,2) in the next line will cause this value of UND$ to be printed centred on the second line of the screen. The main program then prints a blank line and then a comment, with CHR$(33) being used to obtain an exclamation mark.

There is quite a lot to grasp in this first example. The first point is that using the name of a procedure will interrupt the normal flow of a program, allowing something to be done, and then resuming as if nothing had happened to interrupt the flow of the program. The second point is that a procedure call like this allows you to specify variables whose values can be passed to be used by a procedure. Note that I said that their values are passed – you don't use T$ or UND$ in the procedure in the listing above, you pass their values to A$ and N% and then use these variables. This way, one procedure can be called many times and by many different programs, each passing on different data. You aren't forced to pass values if you don't want to, but using this method allows you to write procedures that you can use much more easily and in more than one program; you can keep a stock of procedures for all your needs, and the centring procedure could well be one of them. The third point is that the variables A$ and N% used inside the procedure are local to the procedure. If you put in a line such as:

```
PRINT A$
```

anywhere outside the centre procedure, then nothing will be printed, A$ simply does not exist except inside the procedure. If you want to use other variables inside the procedure and ensure that they also exist only within the procedure, you can use the word LOCAL to label them, such as:

```
LOCAL a,nr,b$
```

You can even have other variables used in the rest of the program with these names, but with entirely different values. A procedure is a little program working in a world of its own, and it can be unaffected by the rest of the program expect for the variable values that are passed to it. In OPL, every program is itself a procedure, with the difference that some can be run independently. You can, for example, run the main program as listed earlier so that it will then call the procedure centre, but you cannot run the procedure centre by itself because it cannot run unless one string and one integer are passed to it.

The word LOCAL is used, then, to show that variables will be used in a procedure (which can mean a main program) and will not exist anywhere else. You can, if you want to, use the word GLOBAL to describe your variables, so that you could have a main procedure starting:

```
GLOBAL A$(20),N%,X%,J%,B$(5,20)
```

with these global variables. What this means is that this program could then call up other procedures which could also make use of these

variables without needing the values to be passed. To show the difference, look at a version of the program ('pro4x2') which uses global variables:

```
pro4x2:
global t$(20),n%
t$="OPL Computing"
n%=1
cen:
t$=rept$("-",len(t$))
n%=2

cen:
print
print"Neat"+chr$(33)
get
cen:
at (21-len(t$))/2,n%
print t$
```

This listing is shown, with variables T$ and N% being declared as global, meaning that they can be used by any other procedure which includes these names. When you want anything centred now, however, it has to be assigned to the variable name of T$, and its line number has to be assigned to N%. By way of compensation, you do not have to put any quantities into brackets when you call the centring procedure.

Normally, we try to avoid using global variables, because their use ties you down to specific variable names which must be used. Global variables can lead to considerable confusion because the same variable names have to be used in both the calling procedure (the main program) and the called procedure.

Another risk is that your programs can become full of lines that are used only to transfer a value from one variable to the one that has to be used by the procedure that will be called, lines like T$=A$ or N%=X%, simply to ensure that these global names will be used. The most damning point against global variables is that they tie down your use of a procedure. When you write a procedure that uses only the variables that are passed to it within the brackets following its name, plus any local variables of its own, then that procedure is a universal one that can be called by any other program or other procedure. If you have a procedure which uses local variables A$ and X%, then this does not prevent you from using A$ and X% as local variable names in any other procedure.

In addition, using local variables avoids the risk of variables being altered 'accidentally'. Suppose, for example, that you have a main procedure which uses a global variable J%. If a procedure is then called which has a line such as

```
J%=2
```

then this will change the value of J% that the main procedure uses. If J% is declared as local in both procedures, then J% can have different values in each. If all variables are local, values are passed from a procedure only by way of quantities within brackets when the procedure is called.

Similarly, a value is passed back from a procedure only by way of a RETURN instruction with a quantity within brackets (see later). Using local variables isolates your procedures against unintended changes in values, and the only price you need to pay is a little more attention to passing the quantities that you need to use.

The place of global variables, as we shall see later, is in a set of procedures that belong only with each other, and in which the use of local variables would cause difficulties, requiring values to be passed to and from procedures too often or requiring more than one value to be returned from a procedure.

A procedure that uses local variables can also pass a value back to the procedure that called it. The following listing ('pro4x3') shows a very simple example of this in action for a conversion routine.

```
pro4x3:
local kg,lb
print"Enter kg."
input kg
```

```
lb=conkg:(kg)
print"=",lb,"lbs."
get
conkg:(n)
local x
x=n*2.2
return x
```

This program uses local variables KG and LB for amounts of kilograms and pounds respectively, and a quantity is assigned to KG by way of an INPUT instruction. The conversion procedure is then called using:

```
LB=conkg:(KG)
```

which means that a quantity will be returned assigned to variable LB after the procedure conkg has been run with the value of KG passed to it. The program then prints out the value of variable LB. In the procedure conkg, which has been separately typed, translated and saved, a local variable X is used in the conversion line X=N*2.2, and then this value is passed back to the calling program by using RETURN X – you can separate the X from the word RETURN by a space or by using brackets.

As usual, because X is local to the procedure its use here will not change any value of a variable X that might exist in the calling procedure (the main program) – in this example the calling procedure has not used variable X. The value is then returned by using RETURN X, and will be assigned to variable LB if the calling line has been written in the form:

```
LB= conkg:(KG)
```

as shown. This does not force you to obtain this value, because if you simply used conkg(KG) then the value would exist but would be ignored. You would hardly want to ignore it in this case, but some procedures might carry out an action that you wanted, like centring a string, but return a quantity that you did not want, like the number of characters in the string. In such a case, you could use the procedure name as if nothing were returned.

Note that many of the functions of OPL work in the same way – you are not obliged to make use of a returned value. Only one value can be returned in this way, so that if you need more than one value to be returned you need to do so by way of global variables that have been declared in the calling procedure.

The use of procedures stored in the memory, then, is something that distinguishes OPL from other versions of BASIC, and makes for very simple and economical programming, since any new procedure that you write can make use of any of the previous procedures that you wrote earlier, providing that they are stored in RAM or in one of the memory packs (EPROM). If you store a procedure in a Datapack, you will need to use its reference letter (B: or C:) when you call it.

Menus

Now for the application to menus, and the following listing ('pro4x4') shows procedures as they might be used as part of a (totally imaginary) games program in which you are invited to choose in the usual Organiser way by selecting a word with the cursor and then pressing the EXE key.

```
pro4x4:
local c%,m%
mnu::
cls
m%=menu("Vampire,Werewolf,Zombie,Mummy,Picket,Quit")
if m%=0 or m%=6:stop
elseif m%=1 :vam:
elseif m%=2 :wer:
elseif m%=3 :zom:
elseif m%=4 :mum:
else m%=5 :zom:
endif
goto mnu::

zom:
print"Routine for"
print"Zombie or picket."
pause -50
```

All of this is placed inside a GOTO loop so that this instruction can be repeated if another choice is wanted following the completion of one choice. The M%=MENU line will assign to the variable M% a number that depends on the position of each word in the menu, using one for Vampire, two for Werewolf and so on. When you see this menu on the screen, it takes up three lines of the screen:

```
Vampire            Werewolf
Zombie             Mummy
Picket             Quit
```

and you need to be careful to avoid menus of this type which would require more than four lines. If you need to have longer menus then you will either have to use the cursor keys to see the hidden lines, or go for a one-line menu in which the menu line can be scrolled sideways. This latter method makes use of the instruction MENUN which is very similar to MENU except that a line number is specified. Note that the words which form the menu choices are placed between one set of quotes in the MENU instruction, not with one pair of quotes per word.

The choice is then carried out by the set of IF and ELSEIF lines that follow. To make these more compact, use has been made of the colon as a separator, so that instead of writing the two lines:

```
IF M%=0 OR M%=6
STOP
```

the simple line:

```
IF M%=0 OR M%=6 :STOP
```

is used – note that there must be a space preceding the colon. The other choices are put into the ELSEIF lines, using the colon as a separator as well as to mark the names of other procedures, and the last of the selection lines uses ELSE since no other choices remain to be made. The selection is ended in the usual way with the ENDIF line, and followed by the GOTO MNU:: which makes the loop endless until the Quit option is chosen from the menu. You could program the endless loop with a DO...UNTIL, using an impossible condition for UNTIL (like UNTIL M%=7), but this is one of the few examples where a GOTO type of loop is preferable. Note that you can use a procedure name more than once if you want to summon the same procedure for different choices.

A procedure is extremely useful in menu choices, but it's even more useful for pieces of program that will be used several times in a program. The centring example is one of these, and others include the searching and sorting routines that we have looked at. The important point is that any one of these routines can be quickly and easily modified to be used with any calling routine that you like to use, simply by passing the correct items to the procedure. For a sorting routine, for example, you would pass the name of the array to be sorted and the number of items in the array.

Rolling Your Own

You can get a lot of worthwhile use and enjoyment from your Organiser when you use it to run programs from program packs that you have bought. You can obtain even more enjoyment from using OPL by typing in programs that you have seen printed in magazines or books. Even more rewarding is modifying one of these programs so that it behaves in a rather different way, making it do what suits you. The pinnacle of satisfaction, as far as computing is concerned, however, is achieved when you design your own programs. These don't have to be masterpieces. Just to have decided what you want, written it as a program, entered it and made it work is enough. It's 100% your own work, and you'll enjoy it all the more for that. The main reason, though, is that a bought program may never allow you to do exactly what you want to do.

Now I can't tell in advance what your interests in programs might be. Some readers might want to design programs that will keep tabs on a stamp collection, a record collection, a set of notes on food preparation or the technical details of vintage steam locomotives. Programs of this type are called 'database' programs, because they need a lot of data

items to be typed in and recorded. There is less reason to need these programs written in OPL than in other varieties of BASIC because you have database facilities built into the Organiser ready to use. On the other hand, you might be interested in the manipulation of numbers or strings in order to deal with your paperwork, adding totals, finding averages, searching for items; all the types of actions that we have been looking at.

What we are going to look at in this section is how a simple program can be designed using procedures because this is a design method that can be used for all types of programs. Once you can design simple programs of this type, you can progress, using the same methods, to design your own programs of any type. The illustration will be of an educational type of program that carries out a quiz action with words – this type has been chosen because it requires no specialist knowledge. You might, of course want to design for yourself a program that calculates radio transmitter ranges, the correlation between pesticide residues and wild-life mortality, one that keeps a record of rainfall and sunshine hours, whatever your specialised needs may be.

The principles of designing such programs are always the same, so that this example provides the same experience as you need for your own purposes with the advantage that its principles are known to every reader, unlike a more specialised program.

Two points are important here. One is that experience counts in this design business. If you make your first efforts at design as simple as possible, you'll learn much more from them. That's because you're more likely to succeed with a simple program first time round. You'll learn more from designing a simple program that works than from an elaborate program that never seems to do what it should. We have already dabbled with the design of simple programs, and I want to show you that this is all you ever need! The second point is that program design has to start with the computer switched off, preferably in another room! The reason is that program design needs planning, and you can't plan properly when you have temptation in the shape of a keyboard in front of you. Get away from it!

Put It On Paper

We start, then, with a pad of paper. For myself, I use a 'student's pad' of A4 which is punched so that I can put sheets into a file. This way, I can keep the sheets tidy, and add to them as I need. I can also throw away any sheets I don't need, which is just as important. Yes, I said sheets! Even a very simple program is probably going to need more than one sheet of paper for its design. If you then go in for more elaborate programs, you may easily find yourself with a couple of dozen sheets of planning and of listing before you get to the keyboard. Just to make the exercise more interesting, I'll take an example of a program, and design it as we go. This will be a very simple program, but it will illustrate all the skills that you need.

Start, then, by writing down what you expect the program to do. You might think that you don't need to do this, because you know what you want, but you'd be surprised. There's an old saying about not being able to see the wood for the trees, and it applies very forcefully to designing programs. If you don't write down what you expect a program to do, it's odds on that the program will never do it!

The reason is that you get so involved in details when you starting writing the lines of BASIC that it's astonishingly easy to forget what it's all for. If you write it down, you'll have a goal to aim for, and that's as important in program design as it is in life. Don't just dash down a few words. Take some time about it, and consider what you want the program to be able to do. If you don't know, you can't program it! What is even more important is that this action of writing down what you expect a program to do gives you a chance to design a properly structured program.

Structured in this sense means that the program is put together in a way that is a logical sequence, so that it is easy to add to, change, or re-design. If you learn to program in this way, your programs will be easy to understand, take less time to get working, and will be easy to extend so that they do more than you intended at first.

As an example, take a look at the numbered list below:

1) Present the name of a country on the screen, picked at random.
2) Ask for its currency unit.
3) Reply must be correctly spelled, using capital first letter.
4) User must not be able to read answer by using EDIT.
5) Allow one mark for each correct answer.
6) Allow two chances at each question.
7) Keep count of number of attempts.
8) Show score as number of correct answers and number of attempts.

This shows a program outline plan for a simple word game. The aim of the game is to become familiar with the names of countries and their units of currency. The program plan shows what I expect of this game. It must present the name of a country, picked at random, on the screen, and then ask what the name of its currency is. A little bit more thought produces some additional points. The name of the currency will have to be correctly spelled.

A little bit of trickery will be needed to prevent the user (son, daughter, brother, or sister) from finding the answers by looking at the program by selecting EDIT and looking for the contents of the arrays. Every game must have some sort of scoring system, so we allow one point for each correct answer. Since spelling is important, perhaps we should allow more than one try at each question. Finally, we should keep track of the number of attempts and the number of correct answers, and present this as the score at the end of each game.

Now this is about as much detail as we need, unless we want to make the game more elaborate. For a first effort, this is quite enough, because if we design it correctly, we can add as much elaboration as we like later. How do we start the design from this point on?

The answer is to design the program in the way that an artist paints a picture or an architect designs a house. That means designing the outlines first, and the details later. The outlines of this program are the steps that make up the sequence of actions. We shall, for example, want to have a title displayed. Give the user time to read this, and then show instructions. There's little doubt that we shall want to do things like assign variable names, dimension arrays, and other such preparation. We then need to play the game. The next thing is to find the score, and then ask the user if another game is wanted. Yes, you have to put it all down on paper! The list below shows what this might look like at this stage.

1) Display title, then instructions.
2) Display name of country.
3) Ask for name of currency unit.
4) Compare reply with correct answer.
5) If correct, increment score, increment tries, ask if another wanted.
6) If not correct, allow another try.
7) If second try not correct increment tries.
8) Ends when N key used in answer to 'Another?' question.

Foundation Stones

Now at last, we can start writing a chunk of program. This will just be a foundation, though. What you must avoid at all costs is filling pages with BASIC lines at this stage. As any builder will tell you, the foundation counts for a lot. Get it right, and you have decided how good the rest of the structure will be.

The main thing you have to avoid now is building a wall before the foundation is complete. It's reasonable to keep most of your instructions and lines of text in the main procedure (the main program) because you will not want to call up these things from any other program. Some of the data manipulations, however, may be useful in other programs and should therefore be carried out by using separate procedures.

This listing ('pro4x5') shows what you should aim for at this stage:

```
pro4x5:
local msg$(20)
msg$="COIN OF THE REALM"
centre:(msg$,1)
pause -30
cls
rem instructions
cls
rem set variables
do
playit:
scorit:
ask:
until done%
```

There are only thirteen lines of program (on screen) here, and that's as much as you want. This is a foundation, remember, not the Empire State Building. It's also a program that is being developed, so we've hung some 'danger – men at work' signs around. These take the form of the lines that start with REM. REM means REMark or REMinder, and any line of a program that starts with REM will be ignored by the computer.

This means that you can type whatever you like following REM, and the point of it all is to allow you to put notes in with the program. These notes will not be printed on the screen when you are using the program, and you will see them only when you Edit.

In the program listing, I have put the REM notes on lines which will later be filled with more instructions. This way, I know where lines need to be added later, allowing me to concentrate for the moment on the more important parts of the program design.

One point that is never obvious is what has to be put into the LOCAL and GLOBAL instructions, because in other versions of BASIC much less forward planning is required – it should be done, but you are not forced to do it. As it happens, this example illustrates the use of global variables rather well, and shows reasons for preferring global to local variables in this case.

Fundamentally, we know that we need to keep a score, so that two numbers will be used, one for the score of correct answers and one of the total attempted. Now we need to assign these numbers in a procedure, but we don't want their values restored to zero each time the procedure runs, otherwise we cannot keep an overall score, only a score for each question. The key here is 'overall scene' – if you have a variable that is concerned with the overall scene, chances are it ought to be a global variable.

Some languages permit you to have local variables that will retain their values between calls, but this cannot be done in OPL. Since both a score and a total number of attempts needs to be updated each time the playit procedure is used, two variables have to be updated each time, and RETURN provides for passing back only one variable value. In this example, then, the use of global variables to keep the score is justified.

Another variable is DONE%. This is going to be used to determine for how long the test goes on, by typing an N answer to a question such as Do you want another?. This variable will be TRUE or FALSE, and we can assign it as being FALSE at the start of the program, altering it only when the n key is pressed at the prompt line of Do you want another?. Once again, making this a global variable allows for simpler design of the procedures, taking into account that these procedures might not be used for other purposes.

Let's get back to the program itself. As you can see, it consists of a set of procedure names for procedures that we haven't written yet. That's intentional. What we want at this point, remember, is foundations. The program follows the original eight point plan of exactly, and the only part that is not committed to a REM or a procedure is the main title – and it uses the procedure centre that was illustrated earlier in this Chapter, and which must be in the memory of your Organiser if this example is to be used.

Take a good long look at this thirteen-line piece of program, because it's important. The use of all the procedures means that we can check this program easily – there isn't much to go wrong with it. We can

now decide in what order we are going to write the procedures and the lines that will replace the REM lines. The wrong order, in practically every example, is the order in which they appear.

Always write the instructions last, because they are the least important to you at this stage. In any case, if you write them too early, it's odds on that you will have some bright ideas about improving the game soon, and you will have to write the instructions all over again.

The next step is to get to the keyboard (at last, at last) and enter this core program. This allows you to test the action of the title at least, though the absence of the other procedures means that nothing else can be tested.

The important point, however, is that this is simple, and it represents the order of actions that you want. A simple core can be built on and elaborated, and you can see from its simplicity whether or not it is carrying out the actions in the order that you want. A complicated core is difficult to follow, difficult to check, and much more likely to give trouble. A simple core is particularly important if you have no printer facilities, because it's very difficult to work on a large piece of program if you can use only the screen of the Organiser.

The next step is to make sure that you have this core program saved and then keep adding to the core. If you have the core recorded, then you can load this back into your Organiser, add one of the new procedures, and then test. When you are satisfied that it works, you can record the whole lot again, using the same filename (unless you want to keep the early versions).

Next time you want to add a procedure, you start with this version, and so on. This way, you keep in the memory a steadily-growing program, with each stage tested and known to work. Again, this is important. Very often, testing takes very much longer than you expect, and it can be a very tedious job when you have a long program to work with. By testing each procedure as you go, you know that you can have confidence in the earlier parts of the program, and you can concentrate on errors in the new sections.

Remember that each separate procedure will be translated and saved independently, and any variables that are local to that procedure are not passed to the main procedure, and will have values that start at zero each time the procedure is called.

First Procedures

The next thing we have to do is to design the procedures. Now some of these may not need much designing. Take, for example, the procedure for the 'Do you want another' question, using procedure ask. This is just a familiar GET$ routine, along with a bit of PRINT, so we can deal with it right away. The listing segment below ('ask:') shows the form it might take, which is not as simple as you might think.

```
ask:
cls
print"Another"+chr$(63)
if get$="y"
done%=0
else done%=-1
endif
```

The variable DONE% from the main procedure has to be assigned as true or false (remember that these are numbers), and because the variable DONE% is global it does not have to be passed to or from the procedure.

If you did not want to make DONE% a global variable, then it would be possible to write the procedure rather differently, perhaps using STOP if the n key were pressed, or passing back a value so that the procedure could be called by using:

```
DONE% = ask:
```

you should try out these possibilities for yourself, because they can lead to this being a more universal routine, one that can be adopted for any purpose that requires something to be printed and a key to be pressed.

Unfortunately, OPL does not recognise the use of the words TRUE and FALSE as being equivalent to the numbers –1 and 0 respectively, so that in this procedure, we have to assign the numbers directly as 0 if the y key has been pressed (not done) and –1 if any other key has been pressed. The test that is used in the main procedure is: UNTIL done%, and this is valid because tests such as WHILE, UNTIL and IF will recognise that –1 means TRUE and 0 means FALSE. Type in the new procedure, using the name ask:, translate it and save it, and now test the core program with this procedure in place.

Now if you simply run the main procedure unmodified, it will stop at the point where the first unwritten procedure is called for. Testing this routine therefore requires a bit of cunning and some modifications of the main procedure. As it exists at present, the main procedure will start its DO loop, and then stop when it encounters the procedure name of playit: because this procedure as yet does not exist.

What we need to do then is to insert a GOTO PLACE:: instruction immediately following DO, and put the PLACE:: label name into position immediately before ask:. This will force the loop to consist of repeating the ask: procedure only, jumping around the unwritten names. You can keep this label system in until the program is almost complete, shifting the position of label PLACE:: each time a new routine is available to test.

Now we come to what you might think is the hardest part of the job – the procedure which carries out the playit action. In fact, you don't have to learn anything new to do this. The playit procedure is designed in exactly the same way as we designed the core program. That means we have to write down what we expect it to do, and then arrange the steps that will carry out the action. If there's anything that seems to need more thought, we can relegate it to a procedure to be dealt with later.

As an example, take a look at the following 'program outline':

1) Keep answers as ASCII codes packed in a string array, three digits per character.

2) Keep questions as another string array.
3) Random number selects both question and answer items.
4) Use variable TR% for number of tries.
5) Use variable SC% for score.
6) Use variable GO% to record number of attempts at one question.

This is a plan for the playit procedure, which also includes information that we shall need for the setting-up steps. The first item is the result of a bit of thought. We wanted, you remember, to be sure that some smart user would not cheat by looking up the answers by selecting to Edit the program. The simplest deterrent is to make the answers in the form of ASCII codes. It won't deter the more skilled, but it will do for starters.

I've decided to put all of the answers in order into string arrays in the form of a string of ASCII codes for each answer, with each code written as a three-figure number. Why three figures? Well, the capital letters will use two figures only, the small letters three, so making them all into three figures simplifies things, because it allows us to move from one letter to another by counting out three figures – what we do is to write a number like 86 as 086, and so on. That's the first item for this procedure.

The next one is that we shall keep the names of the countries in an array. This has several advantages. One of them is that it's beautifully easy to select one at random if we do this. The other is that it also makes it easy to match the answers to the questions. If the questions are items of an array whose subscript numbers are one to 10, then we can place the answers into another string array and make the number that selects the question also select the correct answer. Even neater is to make both questions and answers part of the same array, a two-dimensional array.

The next thing that the plan settles is the names that we shall use for variables. It always helps if we can use names that remind us of what the variables are supposed to represent. In this case we have already

determined that we shall use the global variables of SC% for the score and TR% for the number of tries, so that these are repeated here only as reminders. The third one, GO% is one that we shall use to count how many times one question is attempted. This can be local to this procedure, because the play plan is to allow another go if there is a mis-spelling, and we can set GO% to 0 for the first attempt and to one for a second, allowing no further repetitions after GO% has reached one. This allows us to use a DO loop which contains GO%=GO%+1 and ending with UNTIL GO%=1 OR (some other test). The (some other test) means that there will be a test for a correct answer, but we haven't thought that one out yet. Finally, we decide on a name for the arrays that will hold the country names and the ASCII codes, CNM$ and CNA$. These will have to be added to the global variables in the main procedure.

Play Away

The following listing ('playit:') shows what I've ended up with as a result of the plan:

```
playit:
local choose%,go%,a$(10)
choose%=1+rnd*10
go%=-1
do
  cls
  print"Country is",cnm$(choose%)
  print"Currency is",
  input a$
  ans$=checkit$:(a$,choose%)
  if a$=ans$
  go%=2
  sc%=sc%+1
  else go%=go%+1
  print"Incorrect"
  pause 20
  endif
until go%>=1
tr%=tr%+1
```

The procedure starts by picking a random number in the range one to 10 so that this can be used to pick out the name of the country and also the coded form of the answer.

This random number is assigned to CHOOSE%, which can be a local variable because it need not be used outside this procedure. Another local variable, GO%, is assigned with a value of –1 so that it can be incremented twice when answers are given before it gets to the value of +1.

The next portion is a DO loop which will continue until the value of GO% is one or more. The screen is cleared, and the question is printed. The name of the country is put in by printing the array item which has been picked out by the number CHOOSE%.

The user is then asked for the name of the currency, which has to be typed starting with a capital letter then continuing with lower case letters – failure to use a capital, or using all capitals, will constitute an error. The true answer is then obtained by using the procedure checkit$.

Now we'll look at checkit$ later, and the main thing to note about it at the moment is that it has to have a name that ends with the string sign because it will return a string, the ANS$ that holds the correct answer. The answer can now be checked against the answer supplied by the player, using an IF test. If the answer is correct, the value of GO% is made equal to two, so as to end the loop, and the score is incremented. If the answer is incorrect, using the ELSE clause, variable GO% is incremented, and the message 'Incorrect' is printed, followed by a short pause.

When the loop ends, either because a correct answer has been supplied or because two incorrect answers have been supplied, the variable TR% which holds the number of tries is incremented. This ends the playit procedure which has posed a question, obtained an answer, checked the answer and marked the score. We can now look at the procedure that playit has called, checkit$. The program lines are shown below ('checkit$'):

```
checkit$:(a$,choose%)
local ans$(10),n%,j%
ans$=""
n%=len(cna$(choose%))
j%=1
do
  ans$=ans$+chr$(val(mid$(cna$(choose%),j%,3)))
  j%=j%+3
until j%>n%
return ans$
```

The variable CHOOSE% is the one that we have selected at random, and it's used to select one of the strings of ASCII numbers, CNA$(CHOOSE%). We start by using one local variable N% to hold the total length of the string of digits, and we set another local variable J% to one. A local string, ANS$ is also set to a blank – this could use another name, but it's convenient to keep this name even though it is passed to another variable of the same name in the playit procedure.

Since each ASCII code number consists of three digits, we want to slice this string three digits at a time, and this is done in the DO loop that follows. Starting with J%=1, the string that consists of CNA$(CHOOSE%) has the first three digits extracted by MID$, then converted to number form by VAL, then made into a character using CHR$ and added to the string variable ANS$. This allows the answer string ANS$ to be built up by using this routine with the value of J%, the starting position for slicing, incremented by three on each pass through the loop.

If you find an expression like this hard to follow or to construct, remember that when you have a lot of brackets like this, you read from the innermost set to the outermost. The loop continues until adding three to the value of J% results in a number that is greater than the length of the string which was determined earlier and stored as variable N%.

That's the hard work over. The routine then returns ANS$, and this is an important point. If a procedure is to return a floating point number, then the name of the procedure can be any name that conforms to the rules of eight characters, starting with a letter of the alphabet. If, however, the procedure returns a string, then the name of the procedure must be a string name, ending with the $ sign. A procedure that returns an integer should similarly end with the % sign. Note that the key word here is returns. A procedure can have strings passed to it, and providing that there is no RETURN line that uses a string, the name should not be a string name. It's only when the RETURN line makes use of a string variable name, such as RETURN K$, that the name of the procedure must also carry the string symbol. If you forget this, you will get the FN Argument Error message when you translate the procedure.

Dealing with the Details

We now need to supply some data in order to make the program workable. We need ten strings for array CNM$ and another ten for array CNA$. These are put in by direct assignment, as part of the main procedure as illustrated below :

```
cnm$(1)="Albania"
cnm$(2)="Holland"
cnm$(3)="Greece"
cnm$(4)="Norway"
cnm$(5)="Colombia"
cnm$(6)="Turkey"
cnm$(7)="Malaysia"
cnm$(8)="Indonesia"
cnm$(9)="Pakistan"
cnm$(10)="China"
cna$(1)="076101107"
cna$(2)="071117105108100101114"
cna$(3)="068114097099104109097"
cna$(4)="075114111110101"
cna$(5)="080101115111"
cna$(6)="076105114097"
cna$(7)="082105110103103105116"
cna$(8)="082117112105097104"
cna$(9)="082117112101101"
cna$(10)="082101110109105110098105"
rem set variables
```

- Also add to main routine, following the LOCAL line:

```
global sc%,tr%,done%,cnm$(10,9),cna$(10,24),ans$(10)
```

The CNM$ strings are the names of the countries, and the CNA$ strings are the answers, the units of currency, coded as ASCII numbers of three digits each.

Now there is another procedure to attend to, the scorit procedure which will be responsible for showing the score in terms of the number of correct answers and the number of tries. This is illustrated in program segment below, and is very simple, with no quantities passed in either direction. The score is printed as a fraction like 4/5, and there is a short pause following each display of the score.

```
scorit:
print"Score:",sc%;"/";tr%
pause -30
```

The following segment is the listing for the instructions, a set of straightforward message lines.

```
cls
msg$="You will be shown"
centre:(msg$,1)
msg$="a country name"
centre:(msg$,2)
msg$="and asked to type"
centre:(msg$,3)
msg$="the name of its"
centre:(msg$,4)
pause -50
cls
msg$="currency. Remember"
centre:(msg$,1)
msg$="the capital letter"
centre:(msg$,2)
msg$="and spelling. You"
centre:(msg$,3)
msg$="get two chances."
centre:(msg$,4)
pause -50
cls
```

```
        msg$="The Organiser will"
        centre:(msg$,2)
        msg$="keep the score"
        centre:(msg$,3)
        pause -50
        cls
        rem instructions
```

This shows the instruction lines which are placed on the screen four at a time, using the centre procedure once again. Using centre allows the line number to be chosen, and is a convenient way of allowing four lines to be printed, followed by a pause. You could, if you liked, use a WRITE instruction for this purpose, allowing the instructions to scroll on one line instead of occupying four lines as shown. That's a matter of personal preference, and my preference is to take advantage of the four lines of the LZ machines.

Now you can put it all together, and try it out. Because it has been designed in sections like this, it's easy for you to modify the program. I have deliberately chosen a very simple theme just for this purpose. You can use different data, for example. You can use a lot more data – but remember to change the declaration line to suit. You can make it a question-and-answer game on something entirely different, just by changing the data and the instructions. You can add some sound beeps, for example, or add more interesting display effects.

One major fault of the program is that once an item has been used, it can be picked again, because that's the sort of thing that random choice can cause – you can even find that the same country is picked twice or more in succession. You can get round this by swapping the item that has been picked with the last item (unless it *was* the last item), and then cutting down the number that you can pick from.

For example, if you picked number five, you can swap numbers five and 10, then pick from a total of nine so that the number at the end cannot be picked. This means that the 1+RND*10 step will become 1+RND*D%, where D% starts at 10 (or whatever number you use), and is reduced by one (using D%=D%–1) each time a question has been answered correctly. In this way the game ends when all of the possible

questions have been answered correctly.

There's a lot, in fact, that you can do to make this program into something a lot more interesting. The reason that I have used it as an example is to show what you can design for yourself at this stage. Take this as a sort of BASIC 'construction set' to re-build any way you like. It will give you some idea of the sense of achievement that you can get from working with OPL. As your experience grows, you will then be able to design programs that are very much longer and more elaborate than this one by a long way, and move on to more advanced OPL uses.

5 : Filing Techniques

What is a File?

I shall use the word 'file' from now on to mean a collection of information which we can record in the RAM, on a datapack, or by way of the PC Link on to a disc of another machine. Procedures in the source code of OPL are one type of file, the type known as ASCII files because they consist only of ASCII characters. The other type of file is a binary file, one in which the codes can take a larger range than the ASCII file type, using numbers from 0 to 255 rather than the usual 32 to 127 of ASCII. If there are problems in the recording or replaying of an ASCII file, the result is one or two strange characters or blanks in place of letters, but if a binary file is corrupted it can cause more serious problems. Many data files that are used by the Organiser and by programs such as spreadsheets are binary files, coded so as to decrease the amount of memory that they need to occupy.

In this Chapter, however, I shall use the word 'file' in a narrower sense. I'll take it to mean a collection of data that is separate from an OPL program. For example, if you have a program that deals with your household accounts, you would need a file of items and money amounts. This file is the result of the data-gathering action of the program, and it preserves these amounts for the next time that you use the program. By having the program separate from the data, you can use the same program to work on this month's accounts as was used for last month's, the only difference is that the data is different and it uses a different filename.

Taking another example, suppose that you devised a program which was intended to keep a note of your collection of vintage 78 rpm recordings. The program would require you to enter lots of information about these recordings, such as title, artists, catalogue number, recording company, date of recording, date of issue and so on.

This information is a file, and at some stage in the program, you would have to record this file. Why? Because when you load an OPL main procedure and run it, it starts from scratch. All the information that you fed into it the last time you used it has gone – unless you recorded that information separately. This is the topic that we're dealing with in this Chapter, recording the information that a program uses. The shorter word is 'filing' the information. In this Chapter, we look at the roots of the subject and at the type of filing that OPL BASIC can carry out.

The problem is to avoid duplicating effort. The Organiser comes with a very efficient database program already built into it, and it would be foolish to put a lot of effort into designing a program of your own that could have been created much more quickly by using the database of the Organiser. Programming for yourself is not, or should not be, an exercise for its own sake, and if a ready-made program exists that does the work, then it makes good sense to use it rather than to struggle with something of your own. Be different, by all means, but find a good reason for being different.

Knowing the Names

You can't discuss filing without coming across some words which are always used in connection with filing. The most important of these words are 'record' and 'field', illustrated in Figure 5.1.

File of Friends

Record 1:

Field 1:	Name 1
Field 2:	Address 1a
Field 3:	Address 1b
Field 4:	Birthday 1
(etc.)	

Record 2:

Field 1:	Name 2
Field 2:	Address 2a
Field 3:	Address 2b
Field 4:	Birthday 2
(etc)	

Record 3:

Field 1:	Name 3
Field 2:	Address 3a
Field 3:	Address 3b
Field 4:	Birthday 3

Figure 5.1. The meaning of 'record' and 'field' for a file of data.

A record is a set of facts about one item in the file. For example, if you have a file about vintage steam locomotives, one of your records might be used for each locomotive type. Within that record, you might have wheel formation, designers name, firebox area, working steam pressure, tractive force... and anything else that's relevant. Each of these items is a 'field', an item of the group that makes up a record. Your record might, for example, be the SCOTT class 4-4-0 locomotives. Every different bit of information about the SCOTT class is a field, the whole set of fields is a record, and the SCOTT class is just one record in a file that will include the Gresley Pacifics, the 4-6-0 general purpose locos, and so on.

Take another example, the file 'British Motor-bikes'. In this file, BSA is one record, AJS is another, Norton is another. In each record, you will have fields. These might be capacity, number of cylinders, bore and stroke, gear ratios, suspension system, top speed, acceleration... and whatever else you want to take note of. Filing is fun – if you like arranging things in the right order. The importance of filing is that all of the information can be recovered very quickly, and that it can be arranged in any order, or picked out as you choose. If you have a file on British Motor-bikes, for example, it's easy to get a list of machines in order of cylinder capacity, or in order of power output, or any other order you like. You can also ask for a list of all machines under 250 cc, which ones used four-speed gearboxes, which were vertical twins, which were two-strokes. Rearranging lists and picking out items is something which is a lot less easy when the information exists only on paper.

Filing in RAM or Datapack

In this book, because we are dealing with OPL and its filing instructions systems, we'll ignore anything that you might have learned from using filing programs on other machines, and I'll explain filing from scratch in this Chapter. If it's all familiar to you, please bear with me until I come to something that you haven't met before, because filing on the Organiser is treated in a way that will probably not be completely familiar to you. For a start, you can forget anything you have read about the distinctions between serial files and random access files. These distinctions do not apply to the type of files that OPL deals with, since OPL files are held in memory at all times.

Creating a File

In OPL, a file has to be created before it can be used using the CREATE command. In this sense, creating the file means establishing the filename, a short reference letter for the file, and a list of the types of fields that will exist in each record. This is the information that is needed in order to control the file. Whenever a file is created it will be ready for saving data into, and you can then type the correct variety of data, save it and close the file. When you next need to use the file you do not use the CREATE instruction again, because you do not want to create a new file. The instruction this time is OPEN, meaning that an existing file is to be made ready for reading or writing.

OPL contains functions that allow a procedure to distinguish whether a filename belongs to an existing file or not. This allows you to open a file if the name you type corresponds to an existing file, or create a new file if the name is a new one. The main thing to avoid is trying to create a file which has the same name as an existing file, since this will cause an error message and halt the program. This is considerably

better than wiping out the existing file, but it can be a nuisance. The fact that it can be avoided, because filename detection functions exist, is an excellent feature of OPL.

As always, examples are more useful than descriptions, and since the manual describes only string filing, a spot of number filing in the examples might be useful. One point here is that you can design your own programs so that numbers in files can be worked on, like totalling the numbers in a set of records. The main database program of the Organiser concentrates on sorting and finding rather than on arithmetical work, so that this is an obvious point which makes the use of filing in OPL worthwhile.

The CREATE instruction of OPL needs to specify a full filename, a single-letter reference, and the variable names for up to 16 fields. These variable names can be integer, floating point number or string types, and their appearance in the CREATE instruction is all the declaration that you need – in particular, you do not have to declare the maximum number of characters that might exist in a string within a CREATE instruction. Once the file has been created in this way, values can be assigned to the field variables, and then the instruction APPEND used to add each record to the file.

When you have finished adding data, the file can be closed. This can be done formally, by using a CLOSE instruction, or informally by ending the filing program. When you are dealing with only one file at a time, the formal method is not really necessary, but it's better to get into the habit of closing a file. OPL allows you to juggle with up to four files at a time, and unless you get into the habit of closing files correctly you could run into trouble when you try more advanced work.

```
pro5x1:
create"a:tstrec",a,num,str$
do
cls
print"Item",
input a.str$
print"Value(0 ends)",
input a.num
append
until a.num=0
close
```

The listing above ('pro5x1') illustrates a simple file being created and used. In this file, there will be one string and one floating point number. The string will be used to record the name of an item, and the floating point number to record its value, so that this is the basis for a stock-list, a set of records on income, a note of how sales of goods progressed – any kind of activity in which a name and a number can be used. The important points to look at are how the fields of each record are referred to, and how the file is set up.

The CREATE line is:

```
CREATE"A:TSTREC",A,NUM,STR$
```

of which the name between the quotes is the filename as it will be shown by the Dir command of Utilities – note that this filename will not appear in the Dir list of the Prog action because it is not an OPL file, it is a data file. The letter A within the filename is used to show that the file is in RAM, but if you have datapacks in B or C, then these letters could be used instead. Always use RAM for testing, because once you have recorded on a datapack, the data cannot be erased easily and you can quickly fill a pack with unwanted data.

The next part of the CREATE line is the letter A which this time is a reference letter, a shorthand way of referring to the file. Do not confuse this with the 'drive' letter A:. The purpose of the reference letter (or 'logical' name) is to avoid having to use the full filename of A:TSTREC each time you have to make use of the file. There are four possible letters that can be used here, A to D, because OPL allows you to have up to four files in use at the same time.

If you need to open a fifth file, you will have to close an existing one and re-allocate its reference letter. You can use only one letter at a time (the 'current' file letter), but this letter can be changed, if you have opened more than one file, by the instruction USE. If, for example, you

have opened or created A and B files, then you can change from A to B by the instruction USE B.

The last part of the CREATE line consists of the field names. These are used like variable names but with the advantage that they do not have to be declared separately. This is a particular advantage for a string, because you do not have to declare a maximum length for the string though you are still restricted to a maximum of 255 characters. You can use up to sixteen of these field names and they can be any mixture that you like of floating point, integer and string types, with the names (no more than eight characters) marked in the usual way as for variables.

When the CREATE line runs, the file is created and from that point on it exists in the memory in A:, so that any attempt to create another file of the same name will meet with a warning that the file already exists. You can either use another file name, or make use of the RENAME command to alter the name of the existing file. The remainder of the procedure is now concerned with supplying data to the file and recording that data. No variables are required in this process, so that no LOCAL or GLOBAL declarations are needed in this example – another bonus point for storing information in this format.

The data is then gathered using a DO loop which will be terminated when a zero is entered for the number amount. The screen is cleared, the word Item is printed, and the INPUT step then uses as a variable name a combination of the shorthand name for the file and the field name, in the form A.STR$. The important point here is that this combination can be used just as if it were an ordinary string variable. If you had other files with reference letters B, C and D you could also have INPUTS to B.NAME$, to C.LIST$ or D.title$, assuming that these field names and their files had been created, and you had a USE line preceding each change of file.

The program then asks for a value, a floating point number, which is assigned to A.NUM in the same way. This completes the input of information for this record, and the following instruction is APPEND, which adds this record to the end of the current file. If this is the first record, of course, the 'end' of the file is simply the filename, but subsequent records are then added to the end of the previous record. The alternative to APPEND is UPDATE, which will delete whatever record is currently being used and add your new record to the end of the file.

Following APPEND, the loop ends with its UNTIL clause, so that entering a zero for the amount stops the loop. You do not have to enter any string for the last record, simply press the EXE key. The CLOSE instruction then closes the file so that no further work can be carried out on it until it is opened again.

Totalling a File

A file of the type we have looked at could be used in a variety of different ways. If we only ever wanted to refer to the file, for finding items and looking at records, then there would have been no point in using OPL, because the database or Xfiles facilities of the Organiser would have been more suitable. The point of using OPL is to allow much more flexibility in the use of files, in particular to allow you to carry out actions that are not provided for in these built-in programs. One such action is the totalling of numbers, so that we'll look at this point before anything else.

```
pro5x2:
local tot,t%
open "a:tstrec",a,n,s$
first
do
  tot=tot+a.n
  t%=t%+1
  next
until eof
print"Total", tot
print"in",t%-1,"records."
close
get
```

The procedure for recovering and totalling data is illustrated in in the listing above ('pro5x2'). This time two local variables have to be

declared, TOT to hold the floating point total, and T% to count the number of records, which is the same as the number of different items. The file is then opened, using:

```
OPEN "A:TSTREC",A,N,S$
```

note that only the name of the file need be the same, we can use a different reference letter and different names for the fields. The fields must be in the same order, in this example one float and one string, as they were in the CREATE line.

The next instruction is FIRST, meaning that the first record in the file should be located. The way that filing operates is to keep a 'file pointer', which is a number for a location in memory, assigned to a record, and there are several filing commands whose only action is to move the file pointer. FIRST is one of these instructions, and the others are LAST, NEXT, and BACK. There is also an instruction POSITION which will select a record by its number. By using FIRST here, we ensure that we can look at the whole of the file in order of entry in the DO loop that follows.

In the DO loop, the number variable TOT has the field A.N added to it in order to keep a total of all the numbers in the file. The variable T% is incremented so as to keep a tally of the number of records. Since these two variables were declared as LOCAL, they will both have started with zero values; otherwise they would both have had to be 'initialised' by assigning them with zero. The NEXT instruction then selects the next record, and the end of the loop is detected by using UNTIL EOF, meaning that records will be selected until the end-of-file code is found.

If there is no test for end of file, the selection process will continue, simply reading stray garbage from the memory in place of real data.

The quantities are then printed. The total in variable TOT can be printed as it is, because the last record was the one in which the entered number was 0, which will not upset the total. This last record will, however, upset the record count. Therefore the number of records is decremented so that the displayed number is the number of useful records, not counting the last 'dummy' record which is used to terminate the data entry procedure.

Choice of Use

Very often, it can be useful to keep one comparatively simple data-gathering procedure, but several data-processing procedures, all of which operate on the same information. We can illustrate this by a program which makes use of a menu to allow either the creation or appending of data to a file, along with either the totalling or finding of data, assuming that a simple file consisting of one number and one string per record is being used. For the sake of continuity, so that you can test the procedures on the existing TSTREC file, we will use the same structure.

The procedures are shown below ('pro5x3'):

```
pro5x3:
local m%
do
  m%=menu("ADD,FIND,TOTAL,QUIT")
  if m%=0 or m%=4 :stop
  elseif m%=1 :opit: :addit:
  elseif m%=2 :opit: :findit:
  else m%=3 :opit: :totit:
  endif
until m%=0


addit:
cls
do
  print"Item",
  input a.s$
  print"Cost",
  input a.n
  append
  cls
until a.n=0
last
erase
close
```

```
print"End of entry"
pause 30

findit:
local f%,s$(20)
print"Type word/phrase"
print"-use wildcard."
input s$
first
f%=findw(s$)
if f%=0
print"Not found"
pause 20
else
f%=disp(-1,"")
endif
close

opit:
local nm$(10),g$(1)
print"File name please"
input nm$
if not exist(nm$)
print"No file-create(y/n)"
g$=get$
if g$="y" or g$="Y"
create nm$,a,n,s$
endif
else open nm$,a,n,s$
endif

totit:
local tot,t%
first
do
  tot=tot+a.n
  t%=t%+1
  next
until eof
print"Total", tot
print"in",t%,"records."
close
get
```

Remember that each procedure must be separately translated and saved before it can be used. The main procedure uses M% as a local integer, and in the loop uses this integer to detect the menu choice. The number that has been selected in this way (see Chapter Four) is then used to pick procedures. The IF tests have been written in single lines, using the colon (which must be preceded by one space) to separate instructions, so that if M%=1, then the machine runs procedure opit followed by procedure addit. The colon immediately before the procedure name is the separator, the one immediately following is the usual colon that is part of the procedure name.

Each main menu action starts by opening a file, and if you were writing a program to work on one file for some time you would probably want to organise this differently, with the file being opened before the menu is used and kept open for all the menu actions. In this case, however, the idea is that a different file might be used for each action, and there is no provision for skipping from one action to another with the same file held open. The menu is designed to operate in an endless loop (because of the UNTIL M%=0) so that you escape only by using the QUIT option.

The opit procedure starts with local declarations of a string which will be used for a file name and another which is for a Y/N answer. You are asked to type the file name, and this name is assigned to variable NM$ – note that this could be a full name like A:tstfil or a part name like tstfil – in this latter case, the A: RAM memory is used by default. The name is then tested, using the line:

```
        IF NOT EXIST(NM$)
```

which will return TRUE if the file does not exist, FALSE if it does. If the file does not exist, you are asked if you want to create a new file. This is not done automatically, because you might have mistyped the name and not wish to create a new file. If you answer with a 'y' or 'Y' then CREATE is used with the filename that you supplied. If the file exists, however, the ELSE section of the test will cause the file to be opened ready for use, and the procedure ends.

Procedure opit is used before any of the adding, finding or totalling procedures so as to have a file open, but when finding or totalling are to be used, there would not be much point in creating a new file. Nothing in the program makes this impossible, but the Operating System of OPL will reject the actions on a file which has been created but not used. See Chapter Six for one way of avoiding this problem.

The addit procedure follows very closely the example given on page 123 ('pro5x1'), but with a small refinement. This procedure appends new records, and since every set of records ends with a dummy record, the file could soon contain a number of these dummy records, one for each time the ADD option had been used. In the earlier example, it was assumed that only one dummy existed at the end of the file, and it was dealt with in the totalling program by subtracting one from the number of records read from the file. In addit, the LAST instruction makes the last record the current record, the one that any following instruction will refer to, and then the ERASE instruction is used. This will delete this dummy record, ensuring that only valid records remain in the file, and making it unnecessary to alter the counted number of records in the totit procedure. This procedure is very much as in 'pro5x2' (Page 126) except that the OPEN line is omitted, and the number of records no longer needs to be adjusted.

The findit procedure contains more that is new. OPL has two forms of FIND instruction, of which one requires the name of the item that is to be found typed in full. In this example, we have opted for the other form which will accept a 'wildcard', meaning that part of a name can be omitted and replaced by the '*' sign or the '+' sign. The '+' sign will substitute for one character only, the '*' sign for any number of characters. At the INPUT S$ stage, then, you have to type a name and this name must include '*' or '+' characters. This is an important point, because if the name does not include these characters, then even if you type the name correctly you may find that you do not find the record. Typing a name such as on* will get names such as one, only, ondine and so on. The FIRST instruction is used to get to the start of the file, though this is strictly speaking not necessary. When the FINDW instruction is used, it returns a number in F% which will be zero if the string could not be found, but will otherwise be the number of the record. This does not need to be used in this example, because the FINDW action also makes this record the current record so that it can be printed.

The fact that zero is returned for a string that cannot be found allows for a test to be made, and a phrase printed if this happens, otherwise the record that has been found will be displayed. To display the record, this procedure makes use of the DISP instruction which exists in three forms according to the first number within the brackets, which can be –1, 0 or +1.

When –1 is used, as it is here, the string that follows the number inside the brackets is ignored, and the current record is printed, using one line for each field. The cursor up and down keys can be used to move the cursor over different parts of the record, and the display continues until another key is pressed. When DISP is used with the number +1, the string within the brackets is displayed, and any TAB characters (ASCII 9) cause a new line to be taken on the screen. When the number is 0, the string is ignored and whatever was previously displayed with DISP continues to appear on the screen. The cursor keys operate in the usual way, but any other key will cause the ASCII code for that key to be returned.

In this case, DISP(–1, "") causes the record to be displayed, and pressing any key ends the procedure. By using DISP, it is not necessary to design PRINT lines, nor to use GET to give time for the display to be read. The manual shows an example of DISP used in a loop that will allow the procedure to move on only when a specific key is used, and warns against using PRINT or any other screen printing instruction between the use of DISP with +1 or –1 and its use with 0.

Other Selections

You might wish to create a file in which each record held a date. This would make it reasonable to wish to see a total of the numbers in the file for a specified time interval, perhaps between 010180 and 010190. OPL allows you to program for this type of action with its

comprehensive calendar facilities, which are dealt with in more detail in Chapter Six. For the moment, we'll look only at what is needed to create and use records so that dates are incorporated and used for locating records.

The following listing ('pro5x4') is a simple file creating program:

```
pro5x4:
local d$(8)
create"clnd",a,dt$,csh
do
   do
     print"Entry date as"
     print"ddmmyyyy",
     input d$
  until len(d$)=8
  a.dt$=d$
  print"Amount",
  input a.csh
  append
until a.csh=0
last
erase
close
```

This shows a simple file creating program in which dates can be entered along with a number (cash received on that date, perhaps). This is a very simple procedure, in which there is virtually no testing of the date that is being entered, so that you could quite easily enter 31021989 as a date without challenge. This point of data validation will be tackled in Chapter Six, and for the moment only the bare minimum of checking will be used so as to keep the examples short. A file called CLND is created – the file will be in RAM A: because no 'drive' letter has been used, and its fields are just one string for data and one float for, perhaps, a cash amount.

The main loop starts, and then an inner loop runs in which the current date is assigned to variable D$ and tested to make sure it has the correct length. Assuming that the date has been entered as ddmmyyyy, D$ should contain a total of eight characters. The entry will have to be repeated if eight characters have not been entered. This date string is then assigned to field A.DT$, and the number quantity is asked for. Once this has been input the record is appended in the usual way, with the outer loop ending when the number is zero – this will require the entry of a dummy date like 00000000. This last record is erased at the end of the program, so that whatever is entered is never used in any case.

Now it may be that you would prefer to enter dates in other forms, or to have better validation, but all of these things require a lot of programming and a lot of time to sort out. The whole point of using a hand-held machine like the Organiser is to be able to use short and fast routines, with the minimum of effort, even if this means that you have to be more careful.

No program is foolproof, and a lot of effort spent in validating data is certainly worthwhile if the program is to be used by anyone else. In this case, however, you are the user of the Organiser and the writer of the program, and it's up to you to decide how much foolproofing you need, if any.

This little procedure will write a set of records that contain the date in the form of a string, and a number which might be a cash sum. Note that the routine requires you to enter a date, so that this can be used for 'historical' entries. If each entry needs to carry the date on which it is made, then you do not need a date to be entered because the built-in clock/calendar of the Organiser can be used to provide a date, using the DATIM$ function. This format, however, is not nearly so convenient for our purposes, because in order to use the reading program, we need to have the date in the form of numbers for day, month and year. This is better done by using the DAY, MONTH and YEAR functions, and either recording these numbers or combining them into a string and then recording them.

Now to try a method of extracting data between given dates. The key to this is the function DAYS(D%,M%,Y%) which will return a number for any given date. The number is the number of days since January 1st, 1900, so that these date functions are not useful to you if your needs are for historical records before 1900. By converting each date

into a unique integer, we can test each record to find its date number and sum the numbers only in the records that fall into the limits of time that we have set.

The procedure is shown in below ('pro5x5'):

```
pro5x5:
local d%,m%,y%,n%,x%,z%,tot,num%
print"Start date as"
print"day number",
input d%
print"month number",
input m%
print"year number-4 digits"
input y%
x%=days(d%,m%,y%)
print"End date as"
print"day number",
input d%
print"month number",
input m%
print"year-4 digits"
input y%
z%=days(d%,m%,y%)
open "clnd",a,dt$,csh
first
tot=0 :num%=0
do
  d%=val(left$(a.dt$,2))
  m%=val(mid$(a.dt$,3,2))
  y%=val(right$(a.dt$,4))
  n%=days(d%,m%,y%)
  if n%>=x% and n%<=z%
  tot=tot+a.csh
  num%=num%+1
  endif
  next
until eof
print"Total",tot
print num%,"records"
print"between these dates"
get
```

For both the start date and the end date, the procedure prompts for the entry of a day number, a month number and a year number, and the DAYS function is used to find the day integer. For the start date this is X% and for the end date it is Z%. Once again, there is no checking to find if, for example, Z% is greater than X% as it ought to be. The actions of entering the numbers should really be consigned to another procedure, so that the day integer could be returned from this procedure rather than having the entry lines repeated as they are here.

When the starting date number X% and the ending date number Z% have been calculated, the next step is to open the file that contains the records. The first record is selected and the numbers that will be used for counting are zeroed. The main loop then starts. In this loop, the numbers for day, month and year are extracted from the date string in the record by using LEFT$, MID$ and RIGHT$ for extraction and VAL to convert from string form to number form. These number variables are once again used in a DAYS function to get the number N% for that particular date. This number can now be tested – it should be greater than (or equal to) the first date number and also less than (or equal to) the last date number. If it is, then the number in the record is added to variable TOT and the number of records tally is incremented. The next record is selected, and the loop repeats until the end of the file is found.

Changing a Record

It's not difficult to find how to alter a record in a file. You read the item, print it on screen, and then change the item before re-recording it into the file. As in so many other ways, OPL assists in this process by having a function that is specifically intended for this purpose. The function is EDIT, and it can be used on a string either in variable form or as a record field. Note, however, that this operates on strings only, and this is a good argument for keeping all data that needs to be edited in string form. If actions such as number-totalling are needed, after all, each number can be obtained from the string by using VAL.

Here is an illustration of EDIT in use ('pro5x6'):

```
pro5x6:
open"tstrec",a,n,s$
edit a.s$
update
close
```

An existing file is opened – normally you would input the name of a file and test to find if it existed. The file contains a string and a number, and the string is then edited using the EDIT A.S$ instruction. This presents the contents of the string on the screen, with the cursor on the first letter, so that you can make use of the normal editing methods to change this word. When the EXE key is pressed, this new value of A.S$ exists, but it will not be passed back to the file unless the UPDATE instruction is used. Once this has been executed, the file will be altered to reflect the change to this record. You can see the effect of the alteration by using the procedures in the listing given on page 127 ('pro5x3').

If the number has to be altered, editing is less appropriate because the number can be changed using lines such as:

```
PRINT "New number"
INPUT a.n
```

which could be used following the EDIT instruction and followed by the UPDATE instruction. Nothing in the file is altered until the UPDATE instruction has been carried out. EDIT is intended to allow you to make changes in a string, such as the correction of spelling mistakes, without the need to input the whole of the string again. This type of editing is not needed for a number entry.

Data File Work

One point that has emerged is that the data files which are created by any data filing program in OPL are not listed by the Dir command in the Prog menu. As it happens, you can get a list of your data files from the Utilities menu, but it can be handy to have this facility available in your data-handling programs, such as the skeleton program of 'pro5x4'. OPL offers several functions that make it possible to trace your files by writing procedures that can be added to the menu of any such program.

The next listing ('pro5x7') illustrates this with a procedure that lists the data files that are present in a given 'drive', in this example the A: drive.

```
pro5x7:
local d$(10)
d$=dir$("a:")
print d$
do
  d$=dir$("")
  if d$<>""
  print d$
  endif
until d$=""
get
```

The local variable D$ is declared large enough to hold a file name and the instruction:

```
D$=DIR$("A:")
```

is executed. This will return with the name of the first data file, usually MAIN, in the A: drive. Subsequent files can now be read in a loop, using a blank string in the DIR$ brackets, and the loop end can be detected by the fact that D$ will be blank when no more files remain. The file names are also printed in the loop, and an IF test avoids the printing of the last blank value of D$ by testing for this.

The next step is to develop this with another procedure that can be added so that the number of records in each file will be read. The listing below ('pro5x8') shows the alterations that are needed.

```
pro5x8:
local d$(10),n%
d$=dir$("a:")
print d$,
n%=recrd:(d$)
print n%
do
  d$=dir$("")
  if d$<>""
  print d$,
  n%=recrd:(d$)
  print n%
  endif
until d$=""
get
recrd:(d$)
local c%
open d$,b,n,s$
c%=count
close
return c%
```

An integer variable is added to the LOCAL list, to be used for holding the number of records per file. Following the printing step, the comma holds printing in the same line, and a procedure call to recrd gets the number of records corresponding to the file whose name is stored as the string variable D$. This is done once again in the loop, completing the alterations to the main procedure.

The procedure recrd is also shown. The name of the file is passed to it as D$, and the same variable name has been used in the procedure, along with a local C% for the number of records. The file is opened (you need to know the form of the file in order to be able to specify the correct type and number of fields), and the line:

```
C%=COUNT
```

will count the number of records in the field, assuming that the file is of the format described in the OPEN instruction. The file is then closed, and the number stored as C% is returned to the calling procedure.

The procedure for listing the files is universally useful, but the record counting method is not. If a file contains more records than is allowed for in the OPEN instruction in recrd then the system falls down, so that the counting of records has to be done with care.

It is quite possible that all of your data files will use the same record structure, in which case this procedure can be useful to you, but if this is not so, then the use of this procedure should be confined to programs that are intended to handle a specific type of file.

Statistical Work with Files

The Organiser possesses a splendid set of statistical functions, the list functions, and one of the minor drawbacks of OPL is that these list functions cannot be used directly on file fields, only on the contents of listed variables or on floating point arrays. These functions are so useful for the type of work that the Organiser is used for that ways of making use of list functions with files are of considerable practical interest. The technique consists of reading the files into an array and then using the list functions on the array.

Suppose, for example, that we want to find the average and standard deviation of a set of classmarks, a set of measurements on goods-inwards, or a set of readings taken of goods-outwards. The measurements have to be made at one time, the statistical work at another, so that one procedure reads the figures, another one analyses the results. In practice, of course, one main procedure with a menu would call up whatever else was to be used. For this simpler example, the following listing ('pro5x9') illustrates the data gathering portion, which stores the numbers in float form even if integers are used.

```
pro5x9:
local n%
if not exist ("marks")
create"marks",a,nam$,mrk
else
open "marks",a,nam$,mrk
endif
cls
print"Zero to end"
n%=0
do
  print"Name",
  input a.nam$
  print"Mark",
  input a.mrk
  append
  print chr$(15)
```

```
  print chr$(22)
  at 1,2
until a.mrk=0
last
erase
n%=count
cls
print n%,"entries."
close
```

This program has started in the conventional way, using an IF NOT EXIST test to create or open a file with the name MARKS, consisting of a string and a float. The screen is cleared and the phrase 'Zero to end' is printed as a reminder that entering a zero will end the input. This phrase will remain on screen, with only the lines that have been used for entering data cleared on each entry. In the loop, the name is requested and assigned to the string field as usual, and the number is assigned to the number field.

Once the data has been appended to the file, the second and third lines of the screen are cleared by the PRINT CHR$(15) and PRINT CHR$(22) commands respectively. The cursor then has to be replaced on the second line ready for the next input, and this is done using the AT instruction. The loop continues in the usual way until a zero is entered for the number, and then the last record is erased and the number of records counted and displayed. The file is then closed.

Now to make use of the numbers that have been entered, as distinct from printing names and numbers, or finding records, we need to read the numbers into a floating point array in order to make use of the statistical list functions. This is illustrated in the next listing ('pro5x10') which, shows the full range of these list functions used.

```
pro5x10:
local x(100),n%
open"marks",a,s$,n
first
n%=1
do
  x(n%)=a.n
  n%=n%+1
  next
until eof
cls
n%=n%-1
print n%,"marks"
print"Average",mean(x(),n%)
print"Max.",max(x(),n%)
print"Min.",min(x(),n%)
get
cls
print"S.D.",std(x(),n%)
print"Sum",sum(x(),n%)
print"Var.",var(x(),n%)
get
```

One problem is the size of the array. There is no way of knowing in advance how many records may be stored in a file, so that we cannot know how to dimension the array. One solution that looks attractive is to make use of the number N% which has been obtained in listing 'pro5x9' (page 138) as the record count. If the reading routine could make use of this number, passed to it in the procedure name, then it might be possible to use this to dimension the array.

Unfortunately, the LOCAL instruction accepts only numbers for array sizes, not variables, and the procedure will not translate. The only way that this number can be used, then, is to print a warning that too many records exist to deal with.

Some practical experience, however, helps here. If the routine is being used for classmarks, we should have some idea of the maximum possible number, and the same should be true for the number of goods-inwards or goods-outwards that can be tested in a day, the number of complaints about the service, and so on.

This ought to allow the array to be dimensioned generously enough so as to be most unlikely to fail to accommodate the number of records that will be read – and we can still pass in the number N% to print a warning if this should not be the case.

The procedure shown above ('pro5x10') declares an array of 100 floating point numbers, along with a variable N% for counting them

in. The file is opened, and set to the first record, with the counter N% set to unity – this is important because N% will be used as the array subscript number, and OPL, unlike most versions of BASIC, does not allow an array member such as X(0); the first member of the array must be X(1). The main DO loop then starts, assigning the value from the record field to the array member, incrementing the value of N% and selecting the next record. This is done until the end of file character is read. The value of N% then has to be decremented, as it will have been incremented after reading the end of file character.

The screen is then cleared, and the list functions are used to work on the array. The value of N% is used to show the number of items, and then the average (mean), maximum, minimum are printed. This is followed by a GET to keep these values on screen for reading, and then pressing any key allows the screen to be cleared so that the standard deviation, sum and variance figures can be printed. This ends the procedure.

Last Word

The file handling of OPL allows for very rapid processing, because it is all done in the memory unlike disc-bound programs on desk-top computers. This chapter should have demonstrated how the built-in data processing abilities of the Organiser can be extended by comparatively short and simple routines written in OPL. These routines can process numerical data particularly well, and in this sense are considerably superior to the routines in the database, though large data processing programs for desktop machines are usually well equipped also for numerical processing.

6 : Finishing Touches

OPL is a well-designed and very comprehensive programming language which is designed specifically for the Organiser, and the version of OPL which is used on the LZ machines is a more advanced version of the original OPL which allows better use to be made of the screen. In this Chapter, we shall look at an assorted set of OPL instructions which have not so far been required in examples. Some of these are 'bell and whistles' instructions which though sometimes useful are not essential for programming. Others are for more specialised applications and will not be used by the majority of Organiser owners, and some are sets of instructions of which a few have been illustrated along with brief explanations of the others. One such set is the date set.

The Date Functions

Of the date functions, one of the most important is DAYS(D%,M%,Y%) which is used to find the date number, the number of elapsed days since January 1st 1900. This is an integer number and one of its main uses, previously illustrated, is to find the number of days between two dates by subtracting one date number from another. The other date functions fall into two classes, the functions which return a string for names of days and months, and those which return a number. In addition, some functions operate only on the current date, as supplied by the built-in clock of the Organiser, others act to convert a date which is supplied in number or in string form.

The DATIM$ function, along with the SECOND, MINUTE, HOUR, DAY, MONTH and YEAR functions, are used to display quantities which are obtained from the built-in clock. The form of use is illustrated in the following listing ('pro6x1'), with the quantities being printed rather than being assigned:

```
pro6x1:
print datim$
get
cls
print"Year",year
print"Month",month
print"Day",day
print"Hour",hour
pause -50
print"Minute",minute
print"Second",second
get
```

The DATIM$ takes more than one line to print if you use the instruction PRINT DATIM$ as here, so that it is better to use it along with the VIEW instruction, allowing the string to scroll sideways. Since the format is a string, DATIM$ can be assigned to any declared string variable so that it can then be sliced as required – slicing is not a simple task because the names of days and months are not all of the same length and a slicing action requires the use of a DO loop, adding characters in turn to another string until a space is found. This sounds tedious, and the reason it has not been illustrated is that OPL contains functions that make such contortions unnecessary.

DATIM$ is a useful way of getting a printed version of the date and time, but for many purposes, reading these quantities as numbers can be more useful. The instructions that bear the names of date and time units will each return an integer, and in the listing above they have been used to print the number beside a piece of text that shows which number is being printed.

Working with time and date numbers is considerably simpler than working with strings because each number is an integer. This makes it easier to calculate elapsed time, for example, because you cannot subtract one time in string form from another but you can subtract one integer from another.

The next listing ('pro6x2') illustrates this principle with a procedure

which will tell you how many hours and minutes you have to work until 5.00 p.m. – it does have more serious applications, though.

```
fig6x2:
local h%,m%
h%=hour
m%=minute
if h%>=17
print"You are late"
stop
else h%=17-h%
endif
if m%<>0
m%=60-m%
h%=h%-1
endif
print h%;"h ";m%;"min. to go."
get
```

The procedure starts by obtaining the hour and minute numbers from the built-in clock, which uses hour numbers 0 to 24, and assigning these numbers to variables H% and M% respectively. The H% number is tested so that if the time is later or equal to 17 hours, it's too late to ask how long you have to wait. If the hour is earlier, then the expression 17-H% gets the number of whole hours that you need to wait. This number will be one hour too much if the number of minutes is not zero, however.

The second IF test checks for the number of minutes. If this is zero, then the hours and minutes are printed just as they are, but if the minutes figure is not zero then the hour figure is reduced by one and the minutes figure is subtracted from 60 to find the number of minutes to the next hour. With the revised figures of H% and M% now processed, the required time is printed in a single line.

Day and Week

OPL provides a pair of useful functions which will find day and week information in number form from the date also in number form. If this sounds rather puzzling, then look at the following listing ('pro6x3'):

```
pro6x3:
local d$(6),d%,m%,y%
print"Please enter date"
print"as ddmmyy"
do
  input d$
  if len(d$)=6
  break
  else
  print"Incorrect-use ddmmyy"
  endif
until d%
d%=val(left$(d$,2))
m%=val(mid$(d$,3,2))
y%=1900+val(right$(d$,2))
cls
print"Week No. is",
print week(d%,m%,y%)
print"Day No. is",
print dow(d%,m%,y%)
get
```

The date is requested from you in the form ddmmyy, such as 060789 or 121090, and this is assigned as a string so that it can be sliced. As before, only minimal testing of the string is used, enough to ensure that only six characters are typed. This is done with a DO loop in which the test:

```
IF LEN(D$)=6
BREAK
```

is used. This ensures that when the correct length of string has been typed the loop is broken without any need to use the UNTIL test. This test uses D%, a variable that has not been assigned.

With D% unassigned, its value is zero, which the computer takes in an UNTIL test to mean FALSE, so that this is a way of ensuring that the loop is endless until the BREAK instruction is executed. This is a neater way of creating the loop than the use of a GOTO. The ELSE part of the test will print a message if the entry has been of an incorrect length, like 1190.

The slicing is then done, using VAL to convert to day, month and year integers, with 1900 being added to the two digits for the year so as to get the correct Y% number. If you are using this program beyond the year 2000 then you will have to adjust this (after all, the Organiser is a reliable and long-lived machine). The WEEK function is then used with the quantities D%, M% and Y% to get the week number in the range one to 52, and the DOW (Day Of Week) function is used to get the day number in the range one to seven, with Monday=1. The main problem is that you more often need to know what is the date of day one in a specified week number, like week 27, and there is no OPL function, nor any quick and simple procedure that will find this.

There are two other date functions, DAYNAME$ and MONTH$, which will take a day or month number and convert it to the name of the respective day or month, returning a string. These allow you to convert dates in DDMMYY form to a more civilised name form which can be printed by the VIEW instruction, as this program ('pro6x4') illustrates:

```
pro6x4:
local d$(6),k$(25),d%,da%,m%,y%
print"Date as ddmmyy"
input d$
if len(d$)<>6
print"Bad date -"
print"program ending"
pause -30
stop
endif
d%=val(left$(d$,2))
m%=val(mid$(d$,3,2))
y%=1900+val(right$(d$,2))
da%=dow(d%,m%,y%)
k$=dayname$(da%)+" "+fix$(d%,0,2)+" "
+month$(m%)+", "+fix$(y%,0,4)
view(2,k$)
```

The date is typed in using the usual DDMMYY format, and this is assigned to a string variable as before, though the testing is of a different form this time. The string is sliced as before to give integer numbers for the day, month and year, and the DOW function is used to find the day of the week number for that day as integer variable DA%. A long string K$ is now packed with the information that is needed.

The string starts with the name of the day, obtained by using DAYNAME$ on the day of week number. This is followed by a space and then the number itself converted into string form by using FIX$. FIX$ requires the number to be converted, the number of decimal places, and the total number of characters to be supplied within the brackets. In this case, no decimal places are needed, and the length is two characters. After another space, the month name is obtained by using MONTH$ on the month number M%, and following another space, the year number is printed using FIX$ again but with four characters specified this time. The whole string is then printed using VIEW – scrolling will take place only when the day name and the month name are both long.

Error Trapping

The normal routines of OPL will stop a program when an error is detected, displaying an error message. In most cases, when errors are programming errors or logical errors, this should prompt you to correct the fault that is causing the error, but some errors are unavoidable. If you opt to read a file, for example, and the file does not exist then this constitutes an error which will stop the program from running. This type of error is so likely to occur that OPL, in common with several other varieties of BASIC, provides the EXIST test so that a procedure can check whether or not a file exists before trying to open it.

Sometimes this automatic attention to errors can cause problems which are very difficult to avoid by careful programming. This is something that has to be emphasised, because all but a very few errors should be eliminated by good program design. What I mean here by the few remaining errors are errors which arise from something that is typed by the user as an input and which is difficult to check within the program. Clearly, the existence of a file does not come under this

heading so that the problem concerns items like impossible dates (like 310289), trying to open a file that is already open, or attempting to assign too many members of an array. For this sort of contingency, OPL allows for the trapping of errors so that a special error routine runs, with the normal error routines being bypassed.

This bypassing of normal error routines should be done only in exceptional circumstances, and should certainly not be a part of every procedure or set of procedures. One considerable danger is that a program could develop an endless loop if the new error routine did not deal with an unsuspected error. The error would cause the error routine to be called which could not deal with it, and would return to the program, which would then detect the error again and call the error routine, and so on endlessly. The other source of an endless loop is an error routine which itself contains an error. This type of problem can be avoided by ensuring that error trapping is removed whenever an error has been trapped. The most important point is to ensure that a 'Low Battery' error is not ignored nor diverted.

The error routines of OPL fall into two classes. The first set is based on the ONERR instruction, which will be followed by a label name. An instruction such as: ONERR routine:: is equivalent to ON ERROR GOTO routine::, so that when any error is detected, the routine which starts at the label name is run. There are three important rules regarding this routine:

1) It should start with the ONERR OFF instruction to disable any further error trapping (what if there is an error in the error routine?).

2) It should specifically detect and report error number 194, the Low Battery error.

3) It should be able to deal with any type of error that might arise.

The last point is important, because if the error routine is aimed to detect and provide for just one form of error, then operating this routine when another type of error is detected might itself cause an error. OPL uses the function ERR which will return a different code number for each type of error, allowing you to discriminate between errors.

Using error detection can lead to some interesting problems. The next program listing ('pro6x5') shows a routine which is intended to solve a common problem.

```
pro6x5:
local a$(6),x
onerr fixit::
startit::
print"Enter number"
input a$
x=100/val(a$)
print x
get
stop
fixit::
onerr off
print"Not number"
pause 50
cls
goto startit::
```

The program accepts a number in string form, and this number is later used in a division. There are two possible errors. One is that the string you give cannot be converted to a number. This occurs if you forget to switch to number entry or if nothing is entered (ie, the EXE key is pressed without an entry).

The other error is caused by 0 being entered. This is because you can't divide anything by zero. The program contains the line ONERR FIXIT:: which will cause any error to make a diversion to the routine that starts at label FIXIT, situated beyond the end of the program. In this routine, error trapping is turned off, a message is printed, then after a pause the screen is cleared and execution returns to the place where a number can be entered so that you can get it right this time.

Now if you try this out, you will find that it works – but only for the first error. If you make a mistake on the second attempt then you will see the STR TO NUM ERROR message appear because the conversion

using VAL could not be done (this applies to the blank string as well as to a string of letters). If the second error is due to entering a zero, you will get the DIVIDE BY ZERO error message. The reason for the trapping being done once only is that the return is to label STARTIT::, and this has been placed following the ONERR step. In the error-handling routine, ONERR OFF has cancelled the error reporting in case of other errors in the error routine, and because of the positioning of the STARTIT:: label, trapping has not been restored for the second attempt. This is dealt with by shifting the position of the label name.

You will find, however, that if you enter the zero, which is, after all, a valid number, that you get the message 'Not Number' from your own error routine, because all errors are being dealt with by the same routine. In this case, the remedy is simple – change the message to 'Unacceptable Entry'.

Error trapping like this is a powerful way of dealing with problems, but it should not be written in to a program until all of the procedures have been extensively tested so as to eliminate as far as possible any other errors. This does not mean that error trapping should be tacked on to a program in an unplanned way, only that the actual routines should not be added until the program is known to run well otherwise.

Error trapping must never be used as a substitute for good planning, because the ease with which an error trapping routine can get into an endless loop makes good planning essential.

On other machines, endless loops are simple to deal with – you switch off. On the Organiser, however, switching off is no remedy, because the loop will still be there when you switch on again, and the ultimate remedy of disconnecting the battery will result in the total loss of all data that is held in the Organiser.

Testing for error type is done by using the ERR function, which returns a number, and you can also use ERR$ which gives the standard error message for each error type, as it would appear on the screen of the Organiser.

```
pro6x6:
local a$(6),x
startit::
onerr fixit::
print"Enter number"
input a$
x=100/val(a$)
raise 194
print x
get
stop
fixit::
onerr off
if err=194
raise err
elseif err=252
print"Not number"
elseif err=251
print"Zero unacceptable"
endif
pause 50
cls
goto startit::
```

The above listing ('pro6x6') shows a re-designed version of the procedure of the previous one, with the type of error detected, and an additional trap for the Low Battery error. Looking at the error routine of this example, the ONERR OFF is followed by a test for the Low Battery error, using IF ERR = 194 – the list of error numbers is at the end of the OPL programming manual. This is followed by the new instruction, RAISE ERR. The effect of RAISE followed by an error number is to invoke the normal error handling routine of OPL for that error, so that RAISE 194 would stop the program and print the Low Battery warning.

The ELSEIF clauses then deal differently with the two known types of error that can occur here. Error number 252 is caused by VAL operating on a string that does not contain a number, and the report for this is 'Not number'. Error number 251 is a divide by zero error, which will arise when the division step is encountered, and this is dealt with by a different message. Following the ENDIF, the usual pause and screen

clear is done, and the program then returns to the point where error trapping is turned on again. If, as happens in some routines, this could not be done because the return should be to a later point in the procedure, the ONERR FIXIT:: instruction could be placed as the last step in the error handling routine before the GOTO LABEL:: step.

There is another use of RAISE which allows for better testing of routines like this. Some errors are not easy to obtain, and the most obvious one is the Low Battery error, since it's hardly feasible to run a battery down in order to test a routine. By putting RAISE 194 into some part of the procedure, anywhere following the ONERR step, the effect of this error can be simulated, and you can see how your program will deal with it. The effect ought to be to give the error report, end the program, and give you a chance to Edit – though the correct course is to shut down and change the battery in this particular case. Any program that uses error trapping should be tested with RAISE 194 because the effects of not getting this error reported are so serious for an Organiser.

The other error trapping method of OPL uses the keyword TRAP along with one of a limited set of commands, most of which are concerned with files, but also including INPUT. The full list is:

APPEND	BACK	CLOSE
COPY	COPYW	CREATE
DELETE	DELETEW	ERASE
EDIT	FIRST	INPUT
LAST	NEXT	OPEN
POSITION	RENAME	UPDATE
USE		

and this inclusion of INPUT allows for the trapping of the main cause of errors – incorrect input types. TRAP is typed immediately before the instruction that it refers to, and its effect is to allow the next line to be executed as if no error had occurred. This means that the following line must be an IF test that will detect and deal with the error, and the manual gives an example of a routine that will detect when a string entry is made in place of an integer number. This is not a fatal error in the sense that it would stop the program running, because the normal routine of OPL in this case is to prompt for the correct entry, but without any message of explanation. By using the TRAP routine, an explanation can be put in to help an uncertain user.

One very common problem in filing programs is to devise a menu system that allows for various actions – but to ensure that the files are open before a file action is called for. In the examples of Chapter Five, a typical menu would lead to various procedures being run, each preceded by the opit procedure to open the file. This meant that the file had to be closed at the end of each procedure, otherwise running another menu choice on the same file would lead to an error – trying to open a file that was already open.

This is an excellent illustration of how useful TRAP can be, because in such a menu it can be very awkward to have to open and close files, particularly when the filename has to be specified again for each opening. If you want to open a file and then work on it with various menu choices, but still preserve the ability to start with any one of the choices, then you need to have the file opening procedure in each menu choice, but trap the 'File open' error.

The listing below ('pro6x7') shows an earlier procedure set, from 'pro5x3', amended in this way.

```
pro6x7:
local m%
global nm$(10)
do
  m%=menu("ADD,FIND,TOTAL,QUIT")
  if m%=0 or m%=4 :close :stop
  elseif m%=1 :opit: :addit:
  elseif m%=2 :opit: :findit:
  else m%=3 :opit: :totit:
  endif
until m%=0
findit:
local f%,s$(20)
print"Type word/phrase"
print"-use wildcard."
input s$
```

```
first
f%=findw(s$)
if f%=0
print"Not found"
pause 20
else
f%=disp(-1,"")
endif

addit:
cls
do
  print"Item",
  input a.s$
  print"Cost",
  input a.n
  append
  cls
until a.n=0
last
erase
print"End of entry"
pause 30

totit:
local tot,t%
first
do
  tot=tot+a.n
  t%=t%+1
  next
until eof
print"Total",tot
print"in",t%,"records."
get

opit:
local g$(1)
if nm$<>""
goto endit::
endif
print"File name please"
input nm$
```

```
if not exist(nm$)
print"No file-create(y/n)"
g$=get$
if g$="y" or g$="Y"
create nm$,a,n,s$
endif
else trap open nm$,a,n,s$
if err=199
print"in use"
pause 20
endif
endif
endit::
```

The OPEN instruction has been amended to TRAP OPEN, and is followed by a test for error number 199 for FILE IN USE. If this error number has occurred, the message is printed, but the action is allowed to continue. The chance of the error occurring, however, is remote because in the main routine, the name variable NM$ has been made global, so that this can be tested in the opit routine. If the name exists, it is because opit has already been run and this file will be used – this is an example of not relying on an error trap for detecting a possible problem. You might like to alter the opit routine so that it gives you the chance of using either the existing open file or a new file. The existing file should be closed before the new file is opened.

TRAP, unlike ONERR, carries no risk of causing endless loops or of covering up battery failure, because TRAP applies only to a limited list of inputs. Before we leave the topic, however, it's useful to look at a routine ('pro6x8'), as listed below, which will avoid problems with date entry, and which illustrates an important point about error trapping.

```
pro6x8:
global dt$(6)
onerr catch::
entry::
print"Enter date ddmmyy"
input dt$
datit:
get
```

```
stop
catch::
onerr off
if err=194 :raise err :endif
if err=247
print"Impossible date"
print"- try again."
pause 20
onerr catch::
goto entry::
endif
datit:
local d%,m%,y%,x%
d%=val(left$(dt$,2))
m%=val(mid$(dt$,3,2))
y%=1900+val(right$(dt$,2))
x%=dow(d%,m%,y%)
```

So far, when we have entered a date in DDMMYY form, there has been no way of determining if the entry was sensible. This is important because an invalid date applied to a date function will cause an error which will halt the program. The error is FN ARGUMENT, error number 247. If this is detected, the message is printed, error trapping is turned on again, and the GOTO takes control back to the entry point. The date string DT$ is made a global variable to make it easier to transfer to the date testing routine, datit which in this example exists only for the purpose of validating the date.

The other important point is that the ONERR line is in the main procedure, but the error occurs in the procedure datit which is called by the main procedure. Placing an error trap in any procedure affects error trapping everywhere from the time that the ONERR line is executed.

It is not something that is specific to a procedure like a local variable. This means that the error trapping mechanism allows a procedure to be terminated other than by using the word RETURN or by reaching the end of the procedure, and in this case, it allows a new entry to be made and the procedure to be re-entered.

Sound System

The beep of the Organiser, familiar by now, can be controlled by a BEEP instruction, or used as it is by the line PRINT CHR$(16). Beep control uses the instruction BEEP which is followed by two integers. The first integer gives the time of the beep in milliseconds, ie, thousandths of a second (so that using 1000 for this number makes the note last for one second) and the second number determines the pitch of the note. Figure 6.1 shows the numbers which can be used for a range of musical notes, and the listing which follows it shows a use of BEEP in a musical interlude of about the same artistic level as that created by ice-cream vans.

(a)

Note	Number	Note	Number
C	1532.93	F#	1127.20
C#	1491.12	G	1069.14
D	1385.03	G#	1014.69
D#	1316.46	A	962.00
E	1250.14	A#	912.55
F	1187.02	B	864.70

The set of notes starts at piano Middle C. For other notes, use the OPL procedure below, finding the note frequencies from a data book that deals with Sound. Use integer approximations after calculating numbers.

(b)

```
local n,f
do
  print "Frequency",
  input f
  n=921600/(78+2*f)
  print n
until f=0
```

Figure 6.1. The BEEP numbers (a) that determine pitch, shown for an octave from Middle C. Other note numbers can be calculated by using the formula shown in the manual, or using the program in (b).

```
pro6x9:
local x$(44),n%
n%=1
x$="08131059105907281059062008130728065906200728"
do
  beep 250,val(mid$(x$,n%,4))
  pause 1
  n%=n%+4
until n%>44
```

The numbers which create the note pitches are held in a string array, as set of four digits, so that they can be read in turn with a MID$ instruction and converted to number form by VAL. The time of each note is uniform at 0.25 seconds, and a short pause has been put in to separate the notes.

Better effects can be obtained if the note duration is controlled as well as the pitch, but the main use, of drawing attention to something, is fulfilled by almost any sequence of notes. There is no provision for altering the volume of notes, and you would hardly expect such luxuries in what, after all, is only a beep command.

Miscellany

Some of the remaining instructions are of less importance, others less used (though not less useful). The UDG (User Defined Graphic) instruction is well-illustrated in the manual, but you have to remember that each user-defined graphic replaces one of the standard set, and will exist only while the OPL program is running. If you use your Organiser for business purposes you are not very likely to need the UDG provisions, and if you want a machine for games then you can buy a specialised games machine for a fraction of the cost of the Organiser.

The CURSOR ON instruction switches the familiar Organiser cursor on for OPL programs, and CURSOR OFF switches it off again, the normal default. The whole Organiser can be switched off from OPL by using OFF, and an interesting variation on this is OFF X%, which will allow the machine to be switched off for a number of seconds that can range from one to 1800 (30 minutes), with the variable X% containing the number, or the number used directly as in OFF 60.

The keyboard state can also be controlled from OPL, using the KSTAT instruction. KSTAT has to be followed by a number in the range one to four, and the effect is to alter the keyboard so that pressing a key will produce either:

1) lower case alphabet (the normal state, with SHIFT giving a number or symbol),

2) upper case alphabet entry (as when the CAP key has been activated, giving a number or symbol when the SHIFT key is used),

3) number (as when the NUM key has been used, with SHIFT giving upper case letters), or

4) number, with SHIFT producing lower case letters.

The program listed below ('pro6x10') illustrates this briefly, using a different KSTAT number for each (string) entry:

```
pro6x10:
local x$(2),n%
n%=1
do
  kstat n%
  print"Try 2 keys"
  input x$
  print"(press any key)"
  n%=n%+1
  get
until n%>4
```

The CLOCK command is used in the form CLOCK(1) to display the clock in the upper right-hand corner of the screen while an OPL procedure is being run. Note that this makes some of the UDG character numbers unavailable, since they are used for the clock display. The clock is turned off by using CLOCK(0). The instruction can also be used in the form X%=CLOCK(Y%), and the number that is returned to variable X% will show the previous state of the clock, 1 or 0.

RANDOMIZE is followed by a number, and it will use that number as a source of the 'random' numbers that are generated by RND.

```
pro6x11:
local n%
n%=1
randomize 2
do
  print int(rnd*10)
  n%=n%+1
until n%>4
get
```

The procedure above ('pro6x11') uses RANDOMIZE 2 to make the number '2' the 'seed' for RND, and the DO loop than prints four numbers generated at random. If you run this several times, however, you will see that the four numbers are always the same.

This can be useful for test purposes because it allows you to duplicate conditions on each run. If you want more truly random numbers, use RND without RANDOMIZE, or use a different seed number in RANDOMIZE in each run, such as a number obtained from the time.

There exist two functions concerned with memory space. FREE gives the number of bytes left in the work space, of the order of 21000 on my Organiser. SPACE is used to get the amount of free space left in the current file, and the file must be opened before the SPACE command can be used.

Finally, there are functions COPYW and DELETEW which act on any files (not just OPL data files) and which have to be used with considerable care to avoid wiping out your Diary or Notepad files.

Tail End

OPL is a remarkably powerful and useful programming language, giving the Organiser capabilities well beyond the reach of any programmable calculator, and rivalling the facilities on many desktop machines.

The OPL of the LZ machines is enhanced as compared to that of the earlier machines, and it now offers true pocket computing for the programmer, along with a very useful and modern form of syntax. The purpose of this book has been to introduce OPL in an accessible way, with illustrations that show the actions of the instructions better than any description. The rest is up to you – happy programming!

A : The PC Link

The PC Link consists of a cable which will connect the Organiser to a PC desktop (or portable) machine which is fitted with a serial port, along with software which will allow the PC machine to transmit or receive files. The cable supplied is intended for the equivalent of the IBM PC or XT type of machine, and for the AT type of machine a different plug ending is required – see your dealer for a suitable adapter. Exchange of files with other types of machines is possible, but you are very much on your own when you elect to use any machine that is outside the PC mainstream. For that reason, only the exchange of files with the PC will be described here.

The supplied disc of software should be copied, keeping the original in a safe place. If you use a hard-disc PC machine, create a directory with a name such as PCLINK and copy the programs on the disc to this directory. You can also create a subdirectory called DATA or OPL to hold transferred files or files which are to be transferred, so that the program files remain in one directory and data files in another.

If you use a twin-floppy PC machine, place the PC LINK programs disc copy in drive A and a formatted blank disc, the data disc, in drive B. If you use a single drive machine you will be prompted when to swap discs. The important program in the PC Link set is called CL and this program will call others into use as needed, which is why the disc must remain in the A: drive, or the programs in the current directory of a hard disc, while PC Link is being used.

When the Organiser and the PC are connected by the cable, the Organiser will be automatically switched on when the CL program runs. This allows you to use the COMMS menu selection. The first action is setting up, and for transfers of OPL source programs or Organiser files the method is much simpler than you might think from scanning the manual, provided you are operating along the cable and not by way of a telephone line.

In this Appendix, for reasons of space, only the straightforward link using the cable between the two machines will be considered.

To set up ready for use, the following will be needed once only – assuming that the machines are connected and the PC is running the program CL:

1) Select COMMS from the menu – you will get a DEVICE MISSING message if the link is not present.

2) The menu that appears is:

```
Transmit      Receive      Setup
Term          Auto
```

3) Select SETUP. Use the down-arrow key to select PROTOCOL.

4) Now use the right or left arrow key to select PSION.

5) Press MODE, select EXIT.

This is all that you need to ensure connections to a PC machine through the cable link. Once this is done, the transfer of OPL files is done as follows, dealing first with transmission of files. Once again, the machines must be connected and the PC running CL.

1) Select COMMS, then TRANSMIT.

2) Select PROCEDURE from the choice of File or Procedure.

3) Type the name of the procedure as it exists in the RAM of the Organiser. Press EXE.

4) Type the name for the file in the PC. If this is to be on the B disc, precede the name with B:. If this is to be on the OPL directory of a hard disc, precede the name with OPL/.

   Examples: B:OPL1 OPL/OPL1

5) Press EXE. The file will transfer, with a loud beep at the finish. The screen message on the PC will alter at times during the transfer, returning to normal at the end.

The transmission of files from the Notepad (those without passwords) or the Diary follows the same pattern, but the FILE option rather than the PROCEDURE option is taken in the small menu that appears when you select TRANSMIT.

To receive a file, ensure that the link is connected and the PC software running.

1) Select RECEIVE, and specify PROCEDURE.

2) Type the name of the procedure as you want it in the Organiser not as it exists in the PC.

3) Type the name of the PC file, which might include a drive letter and/or directory path.

4) Press the EXE key, and the file, if it can be found, will be transferred.

Note that if a file of the same name already exists in the Organiser, you will be asked if you want to delete this file first. There will be no transfer unless the file is deleted or you return to the command and use a different name. As before, the other types of Organiser files can be transferred by using the FILE option rather then the PROCEDURE option.

B: Boolean Actions

The logic actions of AND, OR and NOT appear at first sight to have very odd effects when used on numbers. The effects can be understood only if you know how numbers are stored in binary form, and if this is a closed book to you, then I suggest that you look at a book on machine code programming – though you are unlikely to find a book that deals with the machine code of the Organiser. The important point as far as we are concerned here is that OPL stores all numbers in binary form, as a set of 0's and 1's, and the logic actions work on these binary numbers. If you type NOT(7) for example, then you are acting on the binary form of seven, which, if stored as an integer, is 0000000000000111. The action of NOT is to change each 1 to 0 and each 0 to 1, so that the result is 1111111111111000. This is the number 66528 in denary, but OPL follows the normal convention that any integer number above 32767 is a negative number, and its true value is obtained by subtracting 65536. This makes the result of the command NOT(7) equal to –8.

C : ASCII Codes

ASCII Codes in Denary and Hex

No.	Hex	Char	No.	Hex	Char
32	20	(space)	80	50	P
33	21	!	81	51	Q
34	22	"	82	52	R
35	23	#	83	53	S
36	24	$	84	54	T
37	25	%	85	55	U
38	26	&	86	56	V
39	27	'	87	57	W
40	28	(	88	58	X
41	29	)	89	59	Y
42	2A	*	90	5A	Z
43	2B	+	91	5B	[
44	2C	,	92	5C	\
45	2D	-	93	5D	]
46	2E	.	94	5E	^
47	2F	/	95	5F	_
48	30	0	96	60	£
49	31	1	97	61	a
50	32	2	98	62	b
51	33	3	99	63	c
52	34	4	100	64	d
53	35	5	101	65	e
54	36	6	102	66	f
55	37	7	103	67	g
56	38	8	104	68	h
57	39	9	105	69	i

No.	Hex	Char	No.	Hex	Char
58	3A	:	106	6A	j
59	3B	;	107	6B	k
60	3C	<	108	6C	l
61	3D	=	109	6D	m
62	3E	>	110	6E	n
63	3F	?	111	6F	o
64	40	@	112	70	p
65	41	A	113	71	q
66	42	B	114	72	r
67	43	C	115	73	s
68	44	D	116	74	t
69	45	E	117	75	u
70	46	F	118	76	v
71	47	G	119	77	w
72	48	H	120	78	x
73	49	I	121	79	y
74	4A	J	122	7A	z
75	4B	K	123	7B	{
76	4C	L	124	7C	|
77	4D	M	125	7D	}
78	4E	N	126	7E	~
79	4F	O	127	7F	

D : Dabhand Guides

Introduction

"Up to the usual high standards we have come to expect from Dabs Press......I for one am eagerly awaiting Dabs Press's next attempt to cut away more swathes of complexity from the software and hardware world."

Just two quotes taken from a swathe of complementary reviews now appearing and covering the ever increasing range of Dabhand Guides.

There follows a list of some of our recent and forthcoming titles. If you are interested in any of these books, details of how to obtain them are given at the end of the list. Currently our range of books covers:

- Z88
- IBM Compatibles
- Amiga
- Archimedes
- BBC Micros
- General books

Z88

Procedure Index

Chapter One

Procedure	Page	Details
pro1x1	26	A simple 'program' for checking the SAVE and LOAD actions.
pro1x2	28	A short arithmetic program that will print results on the screen.
pro1x3	30	Using quote marks to show that characters have to be printed on the screen exactly as typed.
pro1x4	31	Using the PRINT instruction to place words on the screen, taking a new line for each phrase.
pro1x5	32	The effect of semicolons to prevent a new line from being taken.
pro1x6	33	Spacing lines with PRINT, and in this example using CLS (unnecessarily) to clear the screen.
pro1x7	33	The AT function – this makes use of both column and line numbers to position the cursor. You can print anywhere on the screen, and in any order by using AT.

Chapter Two

pro2x1	37	Assignment in action. The letter X has been used in place of a number, and the number which is assigned to X can be changed.

pro2x2	42	Using string variables. These are distinguished by the use of the dollar sign. The string variable 'name' can consist of up to eight characters.
pro2x3	43	String and number variables might look alike when they are printed, but they are different.
pro2x4	45	Concatenating or joining strings to make a frame for a title. This is not the same action as addition.
pro2x5	45	An INPUT to a number variable. The quantity that you type must be a number.
pro2x6	48	Using GET$ to get a single-character answer.
pro2x7	57	Using trigonometrical functions to find the area of a triangle.
pro2x8	58	Using random fractions and converting a random fraction into a random integer.
pro2x9	59	How to use DISP and VIEW instead of PRINT.
pro2x10	61	Illustrating precisions of numbers. A float (floating point number) is usually stored as an approximation.

Chapter Three

pro3x1	66	A very simple loop using GOTO for the first and almost only time here. A test has been put into the loop so that you can stop it by pressing the '0' key.
pro3x2	67	Using the DO...UNTIL loop construction. The two keywords mark the ends of the loop.
pro3x3	68	Using the DO...UNTIL loop for a counted number of repetitions.
pro3x4	70	A countdown program, decrementing and printing the count, and making use of PAUSE for timing.

pro3x5	71	A number totalling program which will halt when a zero is entered.
pro3x6	72	A Heads-or-tails program that makes use of ELSE following IF.
pro3x7	74	A more advanced Heads-or-tails program, which includes scoring.
pro3x8	76	Using a WHILE...ENDWH loop for number-totalling.
pro3x9	77	Incorporating a WHILE...ENDWH loop into a 'mugtrap'.
pro3x10	82	Introducing the action of LEN, a member of the string function family.
pro3x11	83	Using VAL to convert a number-string into a true number – showing once again the differences between strings and numbers.
pro3x12	84	Number-to-string conversions using FIX$, GEN$, and SCI$.
pro3x13	85	Extracting letters from strings by using LEFT$.
pro3x14	86	Using the RIGHT$ slicing action to extract letters from the right-hand side of a string.
pro3x15	87	Slicing from any part of a string, using MID$ with a position number and a number of characters. These numbers can, of course, be replaced by variables.
pro3x16	88	Illustrating the action of LOC to find if one string is contained within another.
pro3x17	89	Using ASC to find the ASCII code for letters. Only one letter at a time can be used.
pro3x18	91	Comparing words to decide on their alphabetical order.

pro5x9	152	A data-gathering procedure which will later be used to provide numbers for use with the list functions.
pro5x10	153	Reading numbers from a file into an array so that the list functions can be used.

Chapter Six

pro6x1	158	Using the DATIM$ function, and the other time functions which make use of the built-in clock of the Organiser.
pro6x2	159	Finding how long it will be to 5.00 pm – purely for the work-weary.
pro6x3	160	A procedure to find week and day numbers for a supplied date.
pro6x4	161	Printing dates in string form, with day and month names.
pro6x5	164	A very simple illustration of ONERR in action.
pro6x6	166	Altering the previous example so as to re-activate error trapping on the second attempt.
pro6x7	168	An amended file-management set of procedures in which errors due to using a file that is already open can be eliminated.
pro6x8	170	Date validation by using ONERR along with a date function.
pro6x9	173	An extended set of beeps forming a 'melody'.
pro6x10	174	Demonstrating the use of KSTAT to alter the status of the keyboard CAPS and NUM keys.
pro6x11	175	Using RANDOMIZE to produce a recurring set of 'random' numbers.

Index

A Dabhand Guide

The LZ models of the well-established Psion Organiser extend the uses of this excellent machine in several directions. The most important change is to a four-line screen, which allows better use of menus, and the introduction of several new features, including International time and telephone code data, a stop-watch action, and the use of a Notepad which can be password protected.

The expansion of the screen size, along with improvements to the OPL programming language, however, now make this a machine which is not only extremely rewarding to program for yourself, but one which has endless uses through the built-in programs, leading to a better understanding of the computer.

Whether your LZ is your first Psion machine or a replacement for an earlier model, you will find here the way to better understanding and complete command of this outstanding miniature computer.

Ian Sinclair is Britain's leading computer author. With over 120 books to his credit he has established himself as perhaps the most authoritive and most readable computer writer of his time, a fact which has led to international acclaim.

£12.95

ISBN 1-870336-92-5

9781870336925