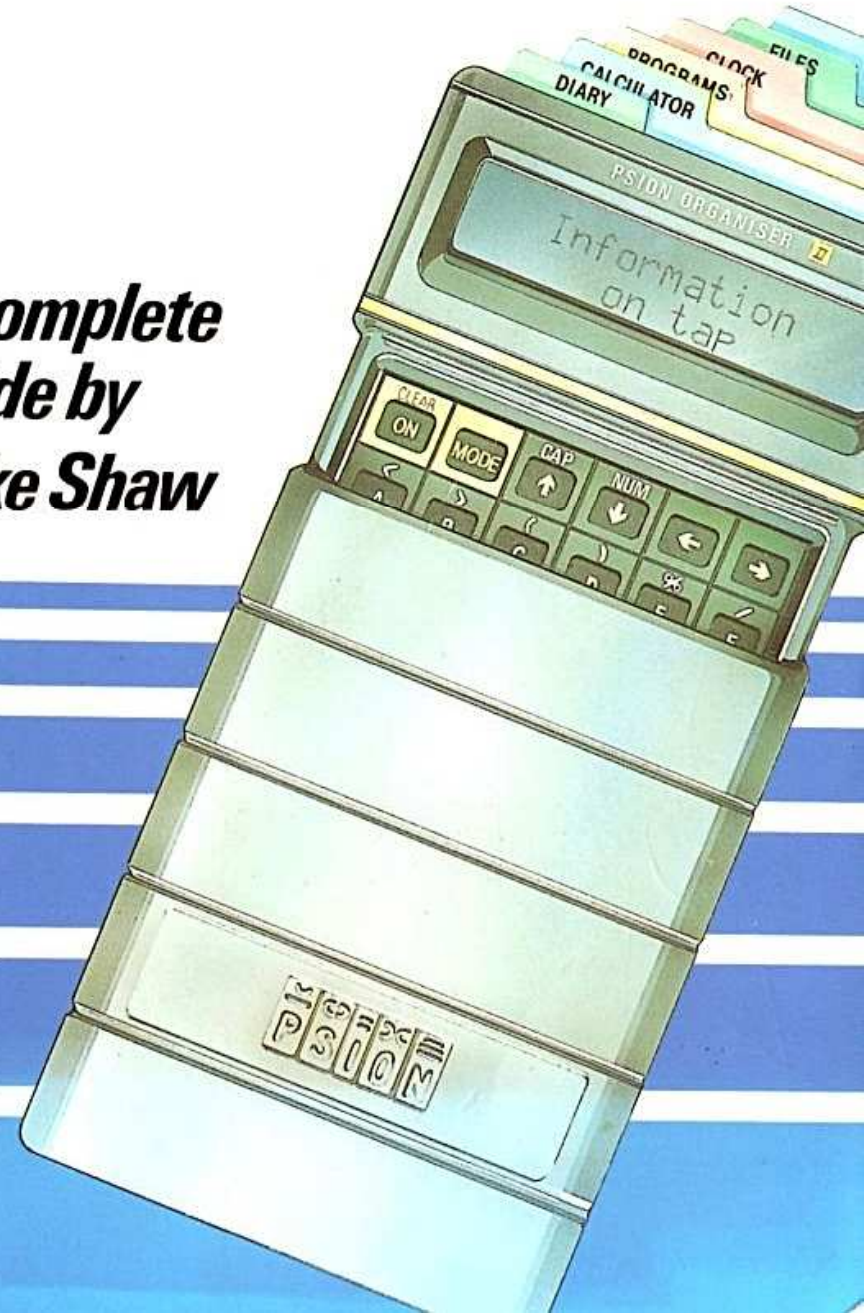
Using and Programming the PSION ORGANISER II



A complete guide by Mike Shaw

Using and Programming Psion Organiser II

A complete guide

USING AND PROGRAMMING PSION ORGANISER II

A complete guide

Mike Shaw

To my lovely Annie, who made it all worthwhile.

Published by
KUMA COMPUTERS LTD.

First Published 1986
2nd Revision August 1987
Kuma Computers Ltd.
Unit 12, Horseshoe Park
Horseshoe Road, Pangbourne,
Berkshire RG8 7JW
Telex 846741 KUMA G Tel 07357 4335

Copyright © 1986 Mike Shaw

Printed in Great Britain

ISBN 07457-0134-5

This book and the programs within are supplied in the belief that its contents are correct and they operate as specified, but Kuma Computers Ltd. (the Company) shall not be liable in any circumstances whatsoever for any direct or indirect loss or damage to property incurred or suffered by the customer or any other person as a result of any fault or defect in goods or services supplied by the company and in no circumstances shall the company be liable for consequential damage or loss of profits (whether or not the possibilty thereof was seperately advised to it or reasonably foreseeable) arising from the use or performance of such goods or services.

ALL RIGHTS RESERVED

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without prior written permission of the author and publisher.

The only exception is the entry of programs contained herein onto a Psion Organiser II for the sole use of the owner of this book.

Acknowledgements

No detailed book about a new product could ever be written without the support of the manufacturer. This book is no exception: I am deeply indebted to all at Psion for the considerable enthusiasm they showed for the project, and for the encouragement, guidance, advice and help they provided during its preparation. I would like to thank especially Martin Stamp for painstakingly reading the manuscript and making invaluable suggestions (and corrections): in so doing, I would not wish to detract from the generous assistance provided by the entire team, without which this book would never have seen the light of day.

CONTENTS

PART 1. HOW ORGANISER WORKS

1.1 The Warehouse Concept, 2
Putting you in the picture; Little boxes;
The busy 'Office'; The power to work;
Keeping in touch; Special duty departments;
Summary.

1.2 The Memory Boxes, 7
More than one type; Look-in *only boxes;*
Put-in look-in *and* change *boxes;*
Put-in, look-in, lock-up *boxes;*
What goes into the boxes; Computers count differently;
Storing 'characters'; Storing instructions;
Storing values; Instruction, character or value?;
Summary.

1.3 Following Instructions, 18
The Jolly Instruction Man; The Translation Service;
Where is the next instruction?; Excuse the interruption;
What would you like to do?; Saving and Finding;
Using the Calculator; Using the other options;
Summary.

1.4 Obeying Program Instructions, 30
Programs and Procedures; Three ways to run a program;
Running a program; Summary.

PART 2. USING THE BUILT-IN APPLICATIONS

2.1 Getting to grips with the keyboard, 36
Not like a calculator;
Selecting CAPS, lower case or NUMbers;
The cursor keys; The CLEAR/ON key; The MODE key;
The DELete key; The EXEcute key; Special keys.

2.2 The main Menu, 41
Switching on; Selecting an option;
Customizing the main Menu; The INFO option;
The RESET option; Switching off.

2.3 **Time and Alarms**, 46
Keeping TIME; Using the ALARMS.

2.4 **Keeping Records**, 49
The built-in filing system; Storing your information; Locating information; Changing record information; Removing records from a file; Copying a complete file; Copying one MAIN file record.

2.5 **Keeping a Diary**, 58
The DIARY Menu; The PAGE option; The LIST option; The FIND option; The GOTO option; The TIDY option; Using the DIARY; Keeping more than one DIARY.

2.6 **Using the Calculator**, 68
Before you start; Fixing the decimal point; Entering a calculation; Very large numbers; The order in which calculations are made; Using the Calculator memories; The built-in functions; Using your own functions; When things go wrong.

PART 3. PROGRAMMING ORGANISER II

3.1 **The principles of programming**, 81
What is programming?; Defining the requirement; Plan the program flow; Summary.

3.2 **Using the program Menu**, 91
The PROGramming options; The NEW option; The EDIT option; The RUN option; The ERASE option; The DIR option; The LIST option.

3.3 **Introducing the programming language**, 101
The types of instruction; Separating instructions; The unknown quantities – variables; Array variables; Choosing variable names; LOCALS, GLOBALs and parameters; Non-declared variables; When things go wrong.

3.4 **From Keyboard to Screen**, 113
Creating a Procedure; Naming the procedure; Declaring numeric variables; Assigning values to variables; Calling one procedure from another.

3.5 **Handling Characters and Strings**, 127
Declaring string variables; Joining strings together; Non-keyboard characters; Cutting the strings; How long is a string?; Where's that string?.

3.6 **Decision making**, 140
Putting things to the test; The Test operators; IF a comparison is true; IF-IF-IF instructions; Using AND and OR to link comparisons; A 'Password' procedure.

3.7 **Creating options and jumping around**, 152
Offering a Menu; Jumping away; The Decorating Materials program.

3.8 **More Keyboard and Screen handling**, 163
Has a key been pressed?; Hang on a moment; Setting up the keyboard; Where's the cursor?; Viewing a string.

3.9 **Looping round until the job is done**, 172
Keep going until–; WHILE a condition is true; DO until a condition is true; Nesting loops; Breaking out of a loop; Making a REMark; Target game program.

3.10 **Converting variables**, 183
A change is needed; Converting a string to a number; Converting numbers to strings; Converting one numeric type to another.

3.11 **Built-in data available on demand**, 190
Time-related information; How much memory is left?; Mathematical values.

3.12 **The sound of music**, 194
Organiser's beep show; Keyboard music-maker; Sound effects.

3.13 **Files – Creating and Opening**, 200
Introduction to files; What goes into a file; The file handling process; Creating a file; Opening a file.

3.14 **Files – writing and changing records**, 207
Putting information on record; Adding a new record; Changing a complete record; See what you are changing; Maximum record size; Selecting which file to use.

3.15 Files – handling records, 214
What record do you want?; Step by step;
Searching for specific information;
Going straight to a record number;
How many records are there?; Viewing your records;
Erasing an unwanted record; Closing a file.

3.16 File Management, 224
Looking after your files; Examine file names;
Datapak free space; Deleting a file;
Copying files; The main File Management procedure.

3.17 Errors and bugs, 232
When your slip shows; The error messages;
Trapping any error;
Trapping specific instruction errors;
When Organiser goes into a never-ending loop;
Finding bugs.

3.18 Machine level instructions, 244
A word of caution; Locating variables in memory;
Examining the contents of memory;
Changing memory contents; Defining your own characters;
Machine language programs.

APPENDIX 253

ASCII – Character patterns

Illustrations

3.1.1 Flowcharts to calculate percentage a smaller number is of a larger number.
3.1.2 Typical Arcade game flowchart.
3.1.3 'Update the Score' flowchart.
3.1.4 Calculation of Decorating Materials flowchart.
3.3.1 The use of GLOBAL variables
3.9.1 Flowchart for 'Target' game.
3.18.1 Character definition grid.

Programs

Note that only *complete* procedures and programs are listed here: for illustrative program sections, see the relevant Chapters.

3.4.1 PF – percentage function.
3.4.2 PF – percentage function, shortened version.
3.4.3 PC – percentage program.
3.4.4 OPF – original price function.
3.5.1 GO – to switch off and 'welcome back'.
3.5.2 MONTH – Month name from its number.
3.5.3 CNTR – Function to centralise a display.
3.5.4 TEST1 – to test CNTR procedure.
3.5.5 MONTHNO% – find month number from its name.
3.5.6 TEST2 – to test MONTHNO% procedure
3.7.1 COW% – Ceiling or walls? (*Decorating program*)
3.7.2 CALEM% – Calculation of Emulsion, long version. (*Decorating program*)
3.7.3 CALEM% – Calculation of Emulsion, short version. (*Decorating program*)
3.7.4 CALPAP% – Calculation of paper. (*Decorating program*)
3.7.5 DECOR – Decorating materials program program.
3.8.1 WAIT – Demonstration of 'PAUSE' (1).
3.8.2 WAIT – Demonstration of 'PAUSE' (2).
3.8.3 KTEST – Keyboard input tester.
3.9.1 MTABLE – Multiplication Table (WHILE/ENDWH).
3.9.2 MTABLE2 – Multiplication Table (DO/UNTIL).
3.9.3 CHARDIS – Display of character patterns.
3.9.4a RND% – Random numbers for Target game.
3.9.4b DELAY – Delays for Target game.
3.9.4c TARGET – Main game procedure.
3.10.1 HEX – Converting decimal to hexadecimal.
3.11.1 TIMER – A simple timing procedure.
3.12.1 BP – BEEP function for frequency inputs.
3.12.2 PLAY – Making Organiser a keyboard instrument.
3.12.3 PHONE – Telephone sound effect.
3.12.4 ALM – Alarm type signal.
3.12.5 SFX – Sound effects sampler.
3.16.1 FDIR – File name Directory.
3.16.2 FSPACE – Memory left on a Datapak.
3.16.3 FDEL – Delete a file.
3.16.4 FREN – Rename a file.
3.16.5 FMAN – File management main procedure.
3.17.1 LOG – Demonstration of ONERR.
3.17.2 TRAPPER – Demonstration of TRAP instruction.
3.18.1a CHRPAT – Character definition control program.
3.18.1b CHRDEF – Defining a character.
3.18.1c CHRSAV – Saving defined character to a file.
3.18.1d CHRLD – Load a pre-saved character pattern.

PREFACE

Mike Shaw's skills include the rare combination of the professional copywriter and the computer buff. He has a remarkable gift for capturing complex or wide concepts and communicating and explaining them with simplicity. As a result, this book can be read with benefit by the expert and uninitiated alike: for the uninitiated, it provides probably the best explanation I have read of how a computer works.

The technology behind Organiser II is a classic example of the application of state-of-the-art microelectronics, resulting in a compact machine that is 'high-powered' yet low on power consumption. I believe that 'hand computers' will become as commonplace as telephones or calculators within a few years. Organiser II is possibly the first of such machines to provide real utility, versatility and breadth, the first *practical* computer to fit the pocket – metaphorically as well as physically.

This book provides an excellent description of that utility and versatility, and illuminates the opportunities made available by Organiser II. It opens the mind wide to the application and potential of the product, while giving an imaginative description of how a computer works and how, in real time, Organiser II performs a multiplicity of different tasks. The author's allegorical figure, JIM, portrays beautifully the huge detail and mechanics of the tasks that the machine must continually perform. One is left with a clear understanding – as well as exhaustion at JIM's immense 'busy-ness'.

The book also takes the reader steadily through the concepts of programming, explaining how to *customize* Organiser II. This is complemented by many fascinating and varied examples of applications – from the calculation of decorating materials to playing music, from file-handling to a simple game – examples which illustrate the possibilities and enhance the understanding of its potential.

Mike Shaw has been involved in the project as a professional communicator from the development phases. His excitement and enthusiasm has led to the creation of this book, which will be an invaluable guide to the many thousands of people who own an Organiser II.

David E. Potter MA (Cantab.) PhD (Lond.)
Chairman, Psion Ltd.
October 1986

PART 1

How Organiser II works

In learning how to use any piece of equipment to its best advantage, it often helps to understand first a little about how it operates.

The aim of this part of the book is to give you an idea of the way Organiser II works - not in great technical detail, but in general terms, by the use of an analogy.

1.1

THE WAREHOUSE CONCEPT

Putting you in the picture

Before discussing how Organiser II tackles all the different functions it provides, let us first create a mental picture of its construction.

You may well have heard some of the language that is associated with computers ... words such as integrated circuits, bytes, processors, RAM and so on.

For the time being, forget them all. Instead, consider the Organiser II not as a computer, but as a large, very busy warehouse that undertakes work on our behalf.

We are going to use this concept to describe its construction and how it works. The description given will, of course, be analogous to the *actual* way Organiser operates. Nevertheless, the picture created will be sufficiently accurate in principle to enable the more important parts of its inner workings to be appreciated.

You may now be asking why you need to understand a little about how Organiser II works before you learn to program it. After all, one can learn to drive a car without knowing how the engine works. The answer is simple. You are not going to learn how to just drive the Organiser – you drive it when you use the built-in functions such as the Diary or Alarms. You are going to learn how to *build in* extra functions to suit your own needs ... you are going to customize the engine to do what you want.

And to do that it helps to know, even roughly, how it works.

Little boxes

So. Organiser II can be considered as a large warehouse. If you look inside, one of the first things you'll notice is that there are thousands upon thousands of small boxes, neatly laid out in an orderly fashion. Each box is individually numbered – rather like the houses in a street. And as with houses in a street, this number is called its 'address'. It enables the box to be located very quickly when the need arises.

The boxes are used to store information for the running of the warehouse, and for handling any work that the warehouse may be asked to do on our behalf. Because they store information, collectively they are called the *memory*. The *size* of the memory is simply the number of boxes available to carry information.

Along the side of the warehouse there are two 'doors', to which can be driven' pantechnicons' containing even more boxes. These

'doors' are the slots in the side of your Organiser II, and the 'pantechnicons' are Datapaks or Program Packs.

There is no limit to the number of Datapaks or Program Packs that can be used with Organiser II, although of course only two can be connected at a time.

In Chapter 1.2, we'll take a closer look at the boxes, to examine the different types used and to see exactly what goes into them. But for the time being, let us continue with our general exploration ...

The busy 'Office'

Tucked away in the heart of the Organiser warehouse there is a small Office, the nerve centre of the entire operation. Very little happens in the warehouse this Office doesn't know about. It controls. It organises (yes, even the Organiser II needs an organiser!). And it handles simple mathematical tasks such as adding or subtracting.

This Office, as you will appreciate, is kept very busy indeed. But in spite of that, it works extremely fast. So fast, it can handle thousands upon thousands of instructions before you could bat an eyelid. Which by any standards, is pretty fast.

The technical name for this Office is *Central Processor Unit*, or CPU for short. For the time being at least, we shall continue to refer to it as the Office. It sounds less daunting.

Unlike its counterpart in most other computer 'warehouses', this Office never closes: as long as it has the power, 'someone' is always on duty.

The power to work

Which brings us to another element in our warehouse: the power supply. Down in the 'basement' there is a battery which provides all the power that's needed. There is also provision, behind a sliding hatch at the top, for connecting Organiser II to a mains supply, using the correct lead.

Even when to all intents and purposes the warehouse is 'off duty' (that is to say, when the Organiser II is switched off), a 'manager' remains on duty. He sees to it that the Clock and Calendar functions, for example, are kept working, so you always have the correct time and date available when you switch on. Also, if you have set Organiser II to give you an Alarm call, he will know about it and act accordingly when the time comes.

The power consumed during these off-duty periods is fairly small. When 'on duty', the activity is greater of course, and the power consumption is higher. If at any time there isn't enough power in the battery to perform 'on duty' work then, via the Office, you will be given the message **LOW BATTERY**. (When you see that message, switch off immediately).

Fortunately, Organiser II has a small back-up reservoir of power.

In the event of the power supply being missing altogether (i.e., if the battery is removed), this reservoir will keep the time and date ticking over for about 30 seconds. This gives you ample time to change the battery without having to reset the time and date again afterwards.

As we shall see, some of the memory boxes inside the Organiser also need a power supply to keep them operational. The back-up reservoir of power will keep these going for about 90 seconds after the battery has been removed. So the chances of losing any of the information contained in these boxes – your valuable information – is made less likely.

However, it is important that notice is taken of the **LOW BATTERY** message: if you don't switch off immediately, there may be insufficient power left in the reservoir to give you time to change the battery. Always switch off before changing the battery.

Keeping in touch

Back to the warehouse concept. All of the memory boxes and the busy Office would be absolutely useless if we were unable to communicate with them. We, after all, are the 'customers' who decide what we want the warehouse to do for us. So we need a way to 'talk' to the Office. This is achieved through the keyboard, used to tap out our instructions.

Similarly, the Office needs to communicate with us – to help us choose what we want to do, to show us what we are doing at the keyboard, and to provide the answers to our requests. The Office's communications appear on the screen above the keyboard. As you will be aware, the screen has two lines and each line can display up to 16 letters, numbers or symbols (such as the '+' and '–' signs). These are called *characters*, and we shall refer to them as such from now on.

If the screen is not large enough to display all the information at one time, the Office arranges for the display to *scroll* from side to side or up and down, allowing us to view the whole message. It is rather like a mask moving over the page of a book, revealing new information as it travels around. We can control the scrolling process by using the special keys marked with arrows on the keyboard: they are called the *cursor* keys.

Generally speaking, the Office lets you know when it is expecting you to communicate with it – that is, enter something at the keyboard – by placing a marker or *cursor* on the screen. The keyboard can be switched to produce 'letters' (i.e. – the characters *on* the keys), or 'numbers' (the characters *above* the keys). When the keyboard is switched to 'letters', the cursor will be a flashing block and an underline character. When the keyboard is switched to 'numbers' the cursor will be just the underline character – no flashing block.

Thus the screen display indicates what type of entry you will be making from the keyboard, and the Office knows exactly what character you want when you press a particular key. The keyboard is discussed in more detail at the beginning of Part 2.

> Try the following. Switch on your Organiser. The screen display will show a range of options: this is called a 'Menu'. Now press the **SHIFT** and **NUM** keys at the same time to switch the keyboard to provide number inputs. The cursor will change to just the 'underline' character – no flashing block.
> Now as you probably know, you can usually select an item on the Menu by simply pressing the key corresponding to the first letter of that item – as an alternative to using the cursor keys to move the cursor to the required item, then pressing the **EXE** key (see Chapter 2.2). So now try to switch Organiser off by pressing the **O** key.
> It doesn't work.
> The Office is expecting you to type a number – or one of the characters printed above the keys. So when you pressed the **O** key, the message received by the Office was 4 – the number printed above the **O** key. At this stage of the game, the Office knows what to do if the first letter of any of the words on its Menu is typed in, but it doesn't know what to do about numbers. So it does nothing – except sit and wait.
> The only way you can select any item from the Menu *while the keyboard is set for numbers* is to use the cursor keys to select the item and then press **EXE**.
> So, to switch off now, either select OFF on the Menu by using the cursor keys and then press **EXE**, or press the **SHIFT** and **NUM** keys at the same time, to reselect letter inputs, then press the **O** key.

Special duty departments

There are several other departments in our warehouse for handling special 'duties'. For example, the Office needs a way to communicate with us audibly, for those occasions when we have 'booked' an Alarm call. There is, therefore, a sound system built in. We can gain access to the sound system – not from the keyboard, but through a program – to control the duration and tone of the sounds produced for our own purposes. This particular aspect of Organiser II is dealt with in detail in Chapter 3.12.

The other 'special' departments need not concern us at this stage: they will be discussed as and when the need arises.

Summary

That completes our preliminary look at the construction of Organiser II. To sum up, it has:

a) Numerous memory boxes, which hold the information Organiser II needs to perform and to remember things for us.
b) A busy Office or *Central Processor Unit* which is in charge of everything that goes on inside the Organiser II.
c) A keyboard to allow us to communicate our requirements to the Office.

d) A screen, so that we can see what we are doing at the keyboard, and so that the Office can communicate with us.
e) Special duty departments – such as the sound system – which help the Office to perform.
f) A power supply, with a small reservoir of power to enable the battery to be changed without losing valuable information.

We will now examine one of the important components – the memory boxes – in a little more detail.

1.2

THE MEMORY BOXES

More than one type

Within the Organiser II there are two different types of memory box. In the Datapaks and the Program Packs, there is a third type. If you use Datapaks, it is important to understand the difference between the type of memory they use and the memories inside the Organiser, in order to save an unnecessary waste of memory space when programming or saving information.

This Chapter discusses the three types of memory box, and then examines in more detail exactly what goes into them.

'Look-in' only boxes

The first and most extensively used type of box inside Organiser II can be *looked into*, to see what it contains, and that is all. Its contents cannot be changed or removed. Even if the battery is disconnected for long periods, the contents of *look-in* only boxes remain intact.

This type of memory box is used to contain information for the running of Organiser II. The Office looks into these boxes, in the main, for the instructions it needs to provide the Diary, Alarms, Clock, Calendar, Record-keeping and Calculator applications, and also for the information it needs to provide the programming facilities.

There are 32,768 boxes of the *look-in* only type in the Organiser II – an indication of just how much memory is needed to provide all these facilties.

Obviously this part of memory cannot be used by us directly: the Office uses it on our behalf whenever we want the Organiser to do something for us.

The process of looking into a box to see what is there is often called *reading*, and consequently these boxes are known collectively as *Read Only Memory* or ROM for short. You have probably heard the word before.

As a matter of interest, the boxes for the *Read Only Memory* in Organiser II have addresses from 32,768 to 65,535.

'Put-in, look-in and change' boxes

We can not only *look into* this type of memory box, we can also *put* information in. If there is an item of information already in the box when something new is put in, then the original item is lost.

It is rather like a cassette tape: when a new recording is made on on it, what was there before is erased. Also, like a cassette tape, this type of box can be 'emptied' so that there is virtually nothing in it at all – its contents are erased.

This type of box is used to store our information on a 'temporary' basis, although 'temporary' can mean as long as we like.

The computer term for a collection of these boxes is *Random Access Memory* or RAM for short. That's because you can *look in*, *put in* and *change* the contents of each box 'at random' or at will.

Within the CM Model of Organiser II there are 8,192 such boxes, while in the XP Model there are 16,384. Most of these boxes are available just for your own use – to keep your records, your Diary information and so on. However, the Office needs some of the boxes for its own 'housekeeping' purposes,
because it too needs to save information on a 'temporary' basis. It cannot use the ROM *look-in* only memory for that.

The small snag with this type of memory box is that it needs a power supply to keep it 'active'. No power supply, no RAM memory. That's why Organiser II has been provided with a small reservoir of power for when you change batteries – to give you adequate time to make the change without losing all your valuable information.

For both Models of Organiser II, the first address of these memory boxes is 8,192. The Office also has another small chunk of RAM all to itself, at addresses from 64 to 255. It will look into the box at address 164, for example, to find out whether the internal sound system should be 'muted' – i.e. should it make a noise for an Alarm call or not. There may be times when you don't want your Organiser to make a noise – when you're at a meeting or the theatre, for example – and by altering the information at this address, you can make your wishes known. (*See your Handbook, Page 151*).

Generally speaking, the boxes at the addresses from 64 to 255 should be used only when you wish to change the way that Organiser II operates – and even then, they should be used with great caution. In any event, they can be accessed only by writing a program.

In practice, you need not worry about the addresses of the memory boxes at all until you become an experienced programmer. Organiser II knows exactly where to put information on your behalf, and it can tell you where that information is, when you really need to know. The important thing is that *it* knows where the information is, and can find it again the moment that you need it. Comforting thought.

'Put-in, look-in, lock-up' boxes

This is the type of memory box to be found in the Datapaks. It combines some of the features of the two memory boxes discussed so far. You can *put* information *in* one of these memory boxes, and you can *look in* to see what is there. But once the information is in, it cannot be changed by Organiser II. So to all intents and purposes, once the information has been put in, it is there for good. Well, almost for good.

If the information that you put into this memory becomes out of date – a friend's address or telephone number changes, for example – then when you enter the new, correct information, you *lock up* the boxes with the old information in, and the new information goes into a new set of boxes. Once the boxes are locked up, you cannot look into them any more.

So what is the benefit of having this type of memory? Quite simply, it doesn't need a power supply. The information the boxes contain remains good and true, even when the Datapak is removed from Organiser II.

This means that there is no limit to the amount of memory you can plug into your Organiser. You could, for example, keep all your personal addresses on one Datapak, and all your business contacts on another. A third could contain details of your stamp or record collection, or your own proven programs. You simply plug in the appropriate Datapak when you want access to the information it contains.

A question arises. If this type of memory gradually gets used up as up-to-date information replaces old, doesn't there come a time when the good information occupies only a comparatively small number of boxes but, because of all the locked-up boxes, there's no room for more information?

Yes. But all is not lost.

The good information can be transferred to a fresh Datapak (or even into the Organiser II's RAM memory, if there's room), and the old Datapak can then be 'wiped' absolutely clean of *everything* that it contains, so that it can be used again as good as new. This 'wiping clean' process is achieved by exposing the Datapak to ultra-violet light (with its protective label removed) for thirty minutes or so. A special unit is available for performing this operation: if your dealer doesn't have one, then Psion will do it for you for a nominal handling charge.

When you plug in a brand new or 'cleaned up' Datapak, Organiser II knows immediately that it *is* completely blank and automatically prepares it to receive information – a process called *sizing*. This simply means organising the Datapak so that the information can be put in correctly on your behalf, and it takes only a few moments. You will see a message on the screen to the effect that sizing is being undertaken.

In computer language, the type of memory we have just been

discussing is called *Erasable Programmable Read Only Memory*, or EPROM for short.

The type of memory to be found in the Program Packs is virtually the same, but of course, you cannot 'lock-up' the boxes: there would be nothing to gain if you could, only everything to lose. The *putting-in* – of program information – has already been done for you. So as far as you are concerned, a Program Pack effectively contains *look-in* boxes only.

What goes into the boxes

As far as we are concerned, all three types of memory box store information the same way.

It would be nice to think that one memory box contains a specific 'chunk' of information, such as an address or a telephone number.

It doesn't.

It doesn't even contain one 'word'.

All that any memory box in Organiser II can hold is a number between 0 and 255. Moreover that number, whatever it is, can represent an *instruction* or part of an instruction, it can represent a *character* such as the letter 'A' or the figure '3', or it can represent part of a *value* – that is, an actual *number* which is to be used for mathematical operations such as adding, subtracting, multiplying, dividing and so on. (The reason why computers need to know the difference between numbers as characters and numbers as values will be made clear later on).

This raises several questions: how can a number represent an instruction, a character or a value – and how does the Organiser know the difference (numbers are, after all, just numbers)? And why is 255 the largest number that a memory box can hold?

Understanding the answers to these questions is fairly important when learning to program the Organiser II. The rest of this Chapter is devoted therefore to providing the answers, starting with why the number stored in any one of Organiser's memory boxes cannot be higher than 255.

Computers count differently

One of the most difficult things for many people to get to grips with during their introduction to computers is the fact that computers have a different way of 'counting'. In our normal, everyday life, we use the *decimal* system of counting. That simply means we have *ten* ' different numbers to count with – 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Computers have only two numbers to count with.

This is because it is far easier to have devices with *two* states than it is to have devices with ten states. Take an ordinary switch, for example. This can be off, or it can be on. If we say that when the switch is off it represents the value '0', and when it is on it

represents the value '1', we have a simple way of representing the values 0 and 1: for '0', we can set the switch to off, and for '1' we can set the switch on.

Not very much use.

However, we can now add a second switch. We can say that when this *second* switch is off it represents '0', and when it is on it represents '2'. Now, by using *both* switches, we can represent the the values 0, 1, 2 and 3. Thus:

The '2' switch	**The '1' switch**		
OFF (=0)	OFF (=0)	represents	0
OFF (=0)	ON (=1)	represents	1
ON (=2)	OFF (=0)	represents	2
ON (=2)	ON (=1)	represents 2+1=	3

Without being too boring, we can add yet a third switch, to represent '0' when it is off, and '4' when it is on. Now, with these three switches, we can represent numbers from '0' to '7'. For numbers '0' to '3', this new switch would be off. Then, for numbers '4' to '7':

'4' switch	**'2' switch**	**'1' switch**		
ON (=4)	OFF	OFF	represents	4
ON (=4)	OFF	ON (=1)	represents 4+1=	5
ON (=4)	ON (=2)	OFF	represents 4+2=	6
ON (=4)	ON (=2)	ON (=1)	represents 4+2+1=	7

We can continue, the same way, until we have eight switches, representing the numbers 128, 64, 32, 16, 8, 4, 2, and 1 respectively when they are *on*. With *all* of the switches *on*, the number they represent is 255 (i.e 128, 64, 32, 16, 8, 4, 2 and 1 all added together).

By setting these switches *on* or *off* in different combinations, we can represent all the numbers between 0 and 255.

(Congratulations. You have just mastered the basics of the *Binary Code*!).

As you will gather, one of the Organiser II's memory boxes effectively contains eight 'switches': the way the 'switches' are set determines the number contained in the box.

But why use only eight switches – why not use ten, say, to allow us to represent even higher numbers? Just as we have seen how *binary* is a convenient counting system for computers to use, so *eight* is also a convenient number in computer technology. In larger computer systems, the memory boxes have 16 or 32 'switches' – always multiples of eight.

Each of the 'switches' in one of our memory boxes is representing a **Bi**nary digit, or **Bit** for short. Hence the Organiser II is called an *eight-Bit* machine.

The block of eight 'switches' in a memory box are called a **byte**.

(Taken together, the first four 'switches' and the last four 'switches' are each called a *nibble*. Don't ask!).

As you know, the abbreviation 'k' (short for 'kilo') usually stands for 1000. In the world of computers, 'k' *actually* stands for 1024. So when you see that a computer has a memory of 8kbytes, it means it has roughly 8000 memory boxes, or, to be precise, it has 8,192 (8x1024) memory boxes.

We need not delve any deeper into the way computers count (sighs of relief).

Storing 'characters'

In Chapter 1.1, we saw that within Organiser II there are several 'Special Departments'. One of these departments is effectively a *library* of patterns. Each pattern in the library is different, and defines the 'shape' of a *character*, as it appears on the screen.

Each pattern is identified by a number: pattern number 65, for example, defines the shape for the capital or upper-case letter 'A', pattern 97 is for the small or lower-case letter 'a', and pattern 51 defines the *figure* '3'. (The Organiser cannot use this pattern number for calculations – and so the *value* '3' has to be stored differently). Within Organiser II's 'library' there are 192 predefined patterns, numbered from 32 to 127 and from 160 to 255 inclusively.

When the number stored inside a memory box represents a *character*, the Office in effect sends a messenger off to the library to look up the corresponding pattern for printing on the screen.

The keyboard of Organiser II allows you to access 79 of these patterns: the upper and lower-case letters, as printed on the keys, the numbers and symbols, as printed above the keys, and **SPACE** – which is really just a blank pattern. When you press a key, it actually passes a *number* to the Office. If the information you are entering at the keyboard is to appear on the screen, then the Office sends off the messenger to look up the corresponding pattern, and that pattern is then reproduced on the screen in the appropriate place.

This happens, for example, when you are entering information that you wish to SAVE as a record.

If you'd like to see what number is associated with each key – and hence with each pattern – do the following:

a) Switch on your Organiser II, and select CALC.
b) Press the **SHIFT** and **NUM** keys together, to select 'letter' inputs. (Your Organiser automatically chooses 'number' inputs when you select CALC).
c) Type in the word **GET**, and then press the **EXE** key. ('GET' is one of the Organiser II's *programming* words which we will come to later. It tells the Office to send a messenger to the keyboard to wait for a key to be pressed, and then to report back the corresponding pattern number. In this instance, it reports the number to the screen).

d) Press any key. The number corresponding to the pattern for the key pressed will appear on the screen. For example, if your Organiser is set for **CAP** letters, pressing the **A** key will cause the number '65' to be printed on the screen. If it is set for lower-case letters, pressing the **A** key will cause the number 97 to be printed on the screen. If you set the keyboard for **NUM**bers, pressing the **A** key will cause the number '60' to be printed on the screen – this being the number for the '<' pattern.
e) To try out another key, press **EXE**, which takes you back to CALC:GET, then press **EXE** again, to execute the instruction, then press the key you wish to see the pattern number for.

If instead of pressing one of the 'character' keys you press one of the 'control' keys (one of the keys marked with an arrow, for example) you will still get a number displayed on the screen. These 'control' key numbers have a special and useful significance when you are writing your own programs: they enable your program to detect, for example, when a specific 'control' key has been pressed.

When you are ready to switch off, simply press the **CLEAR/ON** key two or three times, to return to the main Menu, then press **O**, or select OFF using the cursor keys and press **EXE**.

Earlier, it was stated that there are patterns associated with every number from 32 (the SPACE or blank pattern) to 127 and from 160 to 255. But you can 'get at' only 79 of these directly from the keyboard. You can get any of the others printed on the screen through a *program*. The full set of characters is shown in the Appendix.

What about patterns for the numbers from 0 to 31, and from 128 to 159?

Part of the 'library' is allocated to you, so that you can define your own patterns. You can create up to eight patterns for your own use – and these have numbers from 0 to 7. You need a *program* to enter the patterns you require. But note that the patterns you define are lost whenever the Organiser II is switched off. So if you want to use your own patterns in one of your programs, you must arrange for those patterns to be defined as part of the program, (to save you from entering them afresh each time you use the program). A program to define and save character patterns for subsequent use is given in Chapter 3.18.

This still leaves no apparent patterns for numbers from 8 to 31 and 128 to 159. The reason is that these numbers are reserved for use by the Organiser II. Of particular interest are the 'patterns' associated with the numbers 8 to 15: these are called *control characters*, since they each control, in some way, what happens on the screen. The 'pattern' associated with number 8, for example, moves the cursor on the Organiser's screen one place to the left, and deletes any character on the screen in that position.

The 'pattern' for number 16 is, in effect, an instruction to the Organiser to give a short 'beep' on the sound system.

All the control character pattern numbers can be 'called' from within a program, in the same way that the characters unreachable from the keyboard can be called. The remaining numbers, from 17 to 31 and from 128 to 159, need not concern us at all.

One other point should be made while discussing the patterns associated with numbers. There is an internationally recognised 'code' for the numbers from 0 to 127, called **ASCII**. This stands for 'American Standard Code for Information Interchange'. It ensures that there is some kind of compatability between devices that may be connected together. For example, if you connect your Organiser II to a printer through the optional RS232 connector lead, the printer will produce the same pattern for each number (from 32 to 127) as the Organiser. (What a mess-up if it didn't!). The numbers below 32 are recognised as 'printing control' instructions.

Storing instructions

We have already seen that some of the numbers that are stored in a memory box to represent 'patterns' are, in fact, 'instructions'. Pattern number 16, for example, isn't a pattern at all – but an instruction to give a short beep on the internal sound system.

These instructions are quite different from the second meaning a number can have when stored in a memory box.

Without going too deeply into the subject, the Office works entirely in numbers from 0 to 255. It 'speaks' numbers the same way that we speak words. Its 'language' is purely and simply numbers. An instruction for the Office to do something may consist of one or more numbers.

Each of the numbers in any one specific instruction will be stored in consecutive boxes. In the main, a series of instructions occupy consecutive boxes too.

Take for example two basic instructions, the first of which occupies three boxes. The first box of this instruction could contain a number which tells the Office that it must look at the information held at a specific address, and *hold on to it*. The next two boxes of the instruction will tell the Office what that address is, in its own language.

The next boxes could then contain an instruction which tells the Office what to do with the information it is currently holding. It might be an instruction to display the information on the screen: if this is the case, the information it is holding – a number, remember – is regarded as a *character* and, in crude terms, a messenger is sent off to look at the pattern associated with that number. The pattern so found is then sent off to be displayed on the screen.

This is, of course, an extremely simplistic view of how the Office treats the numbers it finds in the boxes as instructions. Its 'language' contains hundreds of different instructions, most of which occupy several consecutive boxes. Even as programmers, we don't need to know this language – or even understand it any further. Only when you enter the realms of *machine-code programming* do you have to get to grips with what all the numbers mean. The good news is, that's something you need *never* bother to

do. As we shall see later on, the Organiser provides us with a special language, called OPL (*Organiser Programming Language*), to enable us to write programs in a way that *we* can understand.

Storing Values

There will be many times when you want to store or use a number as an actual *value* – when performing a calculation, for example. What's more, you will in no way want to be limited to numbers between 0 and 255 – which is all that can be stored in a memory box.

Values are treated differently from *characters* when they are stored: they have to be stored in such a way that the Organiser can manipulate them to do what you want – add, subtract, multiply, divide and so on.

Consequently, they are stored in a special way, occupying two boxes or eight boxes, depending on the *type* of value being stored.

Whole numbers or values without a decimal point (e.g. 1234) are called *integers*, and always occupy two boxes, even if the number is only '1'. Values *with* a decimal point in them somewhere (e.g. 12.34) are called *floating point* numbers, and always occupy eight boxes.

When you are using Organiser II's calculator, the type of number used is always *floating point* – even though you may be working with whole numbers. The only time you have to be aware of the difference between *floating point* and *integer* numbers is when you are writing programs – and when you are perhaps using those programs while Organiser is in the CALCulator mode.

We will be returning to this subject when we examine 'variables' in the section on Programming.

Instruction, Character or Value?

We have now seen how a memory box contains a number from 0 to 255, and we have seen how that number can represent a character, an instruction (or part of an instruction), or part of a value.

Now, how does the Office know which is which?

By just looking into the box, it doesn't.

It is *told* to regard the contents of a box as a character or part of a value by an *instruction*. A typical instruction, in plain English, might be:

> *Display consecutively on the screen the* ***patterns*** *for the numbers you find in boxes 8554 to 8560.*

Clearly, the numbers in boxes 8554 to 8560 represent characters – they may be the seven letters of a friend's name, or his telephone number. However, should the Office for some reason look into one of the boxes 8554 to 8560 for an *instruction*, then things could go very wrong indeed.

Fortunately, we need not concern ourselves with this problem: the Office is organised enough for such an error not to occur. Only if you program in machine code, or if you misuse *some* of the words in OPL (those which allow you access to the Organiser's operating system), is the problem likely to arise: don't worry, you will be told which are the 'danger' words!

Similarly, the instructions to the Office could have been something like

> *Display on the screen so that we can understand it, the integer value to be found in boxes 8561 and 8562.*

In this instance, the Office knows it must first convert the actual value contained in the boxes to the *characters* for that value, before displaying them on the screen.

Suppose, for example, that the two boxes contain the integer value 4567. The Office converts this value into four 'character' pattern numbers – to give the patterns '4', '5', '6', and '7' – and then it displays these patterns on the screen. So we see, on the screen, the number '4567'. This is the display of a number that can be manipulated mathematically.

Don't panic! The Office handles all this kind of conversion work on your behalf – for floating point numbers as well as integer numbers, so you don't have to worry about it. And as we shall see when we start programming, telling the Office what you want it to do is much easier than the foregoing paragraphs might imply!

Summary

In this Chapter we saw that there are three types of memory box, and that all memory boxes hold numbers between 0 and 255 which can represent instructions, characters or values. To sum up:

The three types of memory box are

a) ROM – *look-in* only boxes whose contents are never lost and cannot be changed. They contain information for the operation of Organiser II.
b) RAM – *put in, look in* and *change* boxes, which hold the information we want to keep, and which need a power supply.
c) EPROM – *put in, look in* and *lock up* boxes, which are to be found in the Datapaks. These boxes don't need a power supply, but the information that's put into them cannot be removed or overwritten *by the Organiser*.

1.2 The Memory Boxes

The numbers contained in memory boxes:

a) Can only be from 0 to 255.
b) Can represent an *instruction* or *part of an instruction*.
c) Can represent a *character pattern number*, to be displayed on the screen, for example.
d) Can represent part of an *integer* or *floating point* value, for mathematical manipulation.
e) The Office assumes that the contents of a memory box represent an instruction or part of an instruction unless it is 'told' otherwise. *You do not have to worry about 'telling' the Office*: the Organiser handles it on your behalf.

1.3

FOLLOWING INSTRUCTIONS

The Jolly Instruction Man

So far, we have seen that the major components used in the operation of Organiser II are memory boxes (by the thousand), an Office, a keyboard and a screen.

During the description of these, brief reference was made to a 'messenger'. This was, of course, an analogy – a convenient way of explaining how information is passed from one place to another within the Organiser. We are going to continue with this analogy, since it makes the way Organiser works easier to understand without getting into deep technical detail.

The 'messenger' is a carrier of instructions and information. He doesn't really exist, but since we have created him, we shall give him a name ... the Jolly Instruction Man, or JIM for short. He travels about inside the Organiser (and into the Datapaks and Program Packs, when necessary), at lightning speed. He gets all of his instructions from the Office, which is in complete control of his actions. When we want the Organiser to do something for us, the Office controls what we want done, and JIM does the running around.

JIM is unique to this book: don't expect to find reference to him elsewhere, and don't expect computer buffs to know anything about him. As mentioned earlier, he has been created simply to help you understand the inner workings of computers in general and Organiser II in particular.

The Translation Service

The kind of instructions that the Office needs are extremely explicit: every small step has to be spelled out in detail. And in the Office's own language (this is called *machine code*).

Happily, we do not have to know this language in order to give the Office instructions – that is to say, to *program* the Organiser. A 'Translation Service' is also built in to the memory system to help us. It's one of the 'Special Departments' mentioned in Chapter 1.1

All we need to learn is the instruction language that the Translation Service understands. This language is called OPL (Organiser Programming Language), and just one 'word' in in it is translated into a whole series of instructions that the Office can understand.

Surprisingly, perhaps, the translated version can take up less

room in the Organiser than the original OPL version – which can be discarded once it is known that the program works to our *complete* satisfaction.

However, it should be noted that the Translation Service works only one way – from the OPL instructions to the Office's own language. The language used by the Office cannot be translated back to OPL. So once the OPL instructions have been discarded, there is no easy way to change the program. The instructions in the Office's own language can also be discarded, of course, and a completely new OPL program written and translated – but it is obviously best to make sure your programs work exactly the way you want before discarding the OPL version.

Since we need to learn the language the Translation Service understands, the way Organiser II works will be discussed in terms of the instructions in that language (OPL), rather than in terms of the language the Office understands.

Where is the next instruction?

Inside Organiser II there are over 32,000 memory boxes containing instructions. These are the ROM boxes. If you have a Program Pack connected, there could be many thousand more boxes containing instructions. And when you write your own programs, there'll be even more boxes just filled with instructions.

The Office needs to know where it is going to get its next instruction from.

Consequently, inside the Office there is a special compartment which contains an *address*. This address tells the Office which memory box holds the next instruction. (Every box is identified by an address number, remember).

This special compartment is called the *Program Counter*: the Office (or CPU) in every computer has a Program Counter.

When power is applied to the Organiser for the first time, the Program Counter has nothing in it – or, to be more precise, it is pointing to the box with an address of zero. The Office immediately sends JIM off to that box, to get the first instruction. From that moment on, things are really buzzing. The sequence of instructions that follow can be regarded as 'housekeeping' – getting the Organiser ready for work. This sequence is performed only when a battery is connected for the first time (or if it is connected after there has been a long break in the supply), and when you RESET the Organiser from the main Menu.

It takes just a fraction of a second.

Ultimately, the Office reaches an instruction which, in effect, tells JIM to go to the keyboard and wait for the **CLEAR/ON** key to be pressed. So far, nothing has appeared on the screen. As far as you are concerned, the Organiser is 'switched off'. As far as the Office is concerned, it now has work to do ... keeping the Clock information 'up to date' for example.

A lot of the work that the Office does goes on 'in the background' – virtually independent of anything we want it to do for us. The Office does this work when the Organiser is switched off as well as when it is switched on.

Before going any further, let us take a brief look at some of these behind-the-scene duties that the Office has to tackle.

Excuse the interruption

Another of the Special Departments referred to in Chapter 1.1 is, in essence, a counting device. It counts hundreds of thousands of times a second at a fixed, precise rate. Whenever it reaches a certain number, it effectively 'rings a bell' in the Office.

It will 'ring the bell', for example, when the counting has been going on for *exactly* one second.

The Office acts on this news immediately. Whatever it is doing at the time, it stops doing it. Even if it is in the middle of an important calculation for you, it stops ... for a split second.

It has been 'interrupted'.

As we have already seen, the address of the next instruction that the Office has to perform is kept in a compartment called the Program Counter. The Office manager takes that address *out* of the compartment, and puts it temporarily in a safe place.

He then puts another address in the compartment, and sends JIM off poste haste to get the instruction to be found in the memory box at that address.

Every time JIM returns with an instruction, the address in the Program Counter box is changed – usually to the next consecutive memory box number. (The instruction JIM brings back could tell the Office to put a completely different address in the Program Counter compartment).

As a result of the instructions brought back to the Office, JIM will be sent all round the Organiser performing specific jobs. In the particular instance currently being discussed, JIM will be sent to the memory boxes containing the latest information about the *time*. He will add 'one' to the contents of the box holding the *second* information and, if necessary, will be told by the Office to also change the minute, the hour, the day of the week, the month and even the year information.

He takes only a few millionths of a second to perform this task.

When he returns to the Office, reporting that all these duties have been done, the Office manager retrieves the instruction address previously put in a safe place, and pops it back into the Program Counter compartment. JIM then carries on where he left off when he was so rudely interrupted ... zooming off to get the next instruction from the specified address.

The result of all this activity is you have the correct time and date information available whenever you want it. And as it all happens so quickly, you're never aware that it's going on in the background.

There are two other 'interruptions' that should be mentioned. The first of these concerns any Alarm calls you may have set – either using the Diary or the Alarm function. Every so often – about ten to twelve minutes – the Office work is interrupted so that JIM can go looking around the Diary and the Alarm memory boxes, to see if an Alarm call is required during the next ten minute period.

If there *is*, JIM reports back to the Office that an Alarm call is required in so many minutes time. The Office makes a note of the fact (in effect), and when the appropriate time comes, again stops what it is doing, saves the contents of the Program Counter, and sends JIM off on another series of errands that ultimately result in providing *you* with the required message. If necessary Organiser II will be switched on, and the sound system will beep to draw your attention to the fact that there is a message on the screen – your Alarm call.

When you press the **CLEAR/ON** key – to clear the Alarm – the Office manager again retrieves the saved address of the next instruction and pops it back in the Program Counter compartment. So if you were in the middle of using Organiser II for something else, you will be returned to where you left off. If Organiser II was off when the Alarm call occured, it switches off again.

The other type of interruption you should know about concerns JIM waiting at the keyboard for something to happen. The Office is well aware of the fact that all the time Organiser II is switched on, it is using more power than when it is switched off.

If JIM waits at the keyboard for more than about five minutes with nothing happening, he gets pretty bored, and assumes you have found something else to do. So he sends a message back to the Office and the Office says, "O.K., let's leave everything as it is, and switch off".

Apart from his forays to up-date the Clock and check for Alarm calls, JIM can now rest by the **CLEAR/ON** key, waiting for you to switch on again. When you do, the Office recalls everything the way it was when it switched off on your behalf, so you can pick up exactly where you left off.

All pretty clever stuff.

What would you like to do?

Let us start by taking a look at what happens when you first switch on Organiser II – the moment JIM has been waiting for at the **CLEAR/ON** key.

As soon as you press this key, JIM reports back to the Office. He is immediately sent off again to collect the main Menu information for printing out on the screen. This Menu information is held in the RAM memory boxes (the *put-in*, *look-in* and *change* boxes), so that you can change it, if you wish, to suit your own purposes.

Once JIM has seen to it that the Menu is on display, he is told by the Office to sit and wait by the keyboard for you to make your selection.

In effect, he is holding up a placard of options, and is saying to you, "Pick one".

He gives you two ways of choosing the option you require.

If you type in a *letter* – be it a capital or lower case letter – he will look through the words on the Menu to see if any of them start with that letter. If he finds just one of the Menu options starting with that letter, he will assume that is the option you want, and immediately arranges (through the Office) for the facility you selected to be provided. If on the other hand there is more than one word on the Menu starting with the letter you select, he will move the cursor to the first of them: each time you press the letter again, he moves the cursor on to the next Menu word beginning with that letter. In these circumstances, he waits for you to press the **EXE** key to select the option you want.

The second way you have of selecting an option is to use the 'arrow' or cursor keys to move the cursor around the screen until it is 'on' the first letter of the option you want. Then, pressing the **EXE** key selects that option.

As mentioned earlier, the words of this main Menu are stored in the RAM boxes, so that you can change them around, add to them or remove them to suit your own purposes. When you write a program, if you want to have that program as one of the options on the main Menu, you can. We'll see how to do this later on. The only word on the main Menu the Office will not allow you to remove is **OFF** because, obviously, without that option you would not be able to switch Organiser II off.

Some of the options on the main Menu, when selected, will provide you with a further set of options. If you select PROG, for example, you will be put into the Organiser II's PROGramming 'mode', and you will be given the choice of writing, editing, testing – and so on – one of your own programs. *These* further Menus, when part of the Organiser's built-in facilities, are stored in the *look-in* only memory (ROM), and cannot be changed or altered in any way.

The Organiser's Programming Language allows you to create Menus for use in your own programs. You could, for example, build up a conversion table – centimetres to inches, litres to gallons and so on – and have the options available on a Menu.

Whenever an option on one of the Organiser's own built-in Menus has been selected, JIM is sent off to examine a special list which gives a memory box address for each option. That address is then brought back to the Office and placed in the Program Counter compartment. If you select the Diary, for example, the address of the first memory box containing the set of instructions associated with the Diary application is popped into the Program Counter

compartment. JIM is then sent off to get those instructions.

When you create a Menu in one of your own programs, things work a little differently. You need to tell JIM what you want him to do when one of your options has been selected – all he will do is return a number relating to the position of the selected option within your Menu. Your program will use that number to decide what JIM must do next. This will be discussed more fully in Part 3 of this book.

Saving and Finding

One of the powerful built-in features of Organiser II is its ability to SAVE information you wish to keep, and to FIND it again with only a small clue as help. This feature involves something called a *file* in the Organiser and, since you can write programs to create your own, separate files, it is worth examining how Organiser performs this task.

A *file* is simply a group of *records*. You can create up to 110 files within your Organiser (and on any Datapak) – memory space permitting – when you write and use programs. However, each file must have its own, individual *name*, to identify it from other files. You may wish to create one file for club members, with details of their addresses and subscriptions, another file for your domestic accounts, yet another for details of a stamp or record collection.

One file is automatically created by the Organiser – so that you can save information straight away, either in the Organiser's RAM memory or on a Datapak, without having to write a program. This file is called 'MAIN'.

A *record* is one 'group' of information within a file. It may be one club member's name and address, his or her club membership number, subscription rate – and whether or not it has been paid, and any other pertinent details.

If you were to write all these details on a card, such as may be found in a card index system, you would probably organise it so that each item of information occupied a specific area on the card. For example, you might put all the subscription information in a bottom corner so that you could quickly flick through and see who has or hasn't paid, or so that you could add up all the subscriptions to see what the total should be.

The records in an Organiser file are arranged in a similar way – the individual 'areas' are called *fields*. And, just as with the index cards, any desired 'field' or 'area' of all or any of the records can be searched and analysed for specific information. All of that is achieved through a program.

Each of the records in the Organiser's MAIN file have just one field. Nevertheless, this file still provides an extremely useful and powerful record-keeping facility: it can be used, for example, to carry all the information you might otherwise keep in an address book – names, addresses, telephone numbers, birthdates, whether or

not the person is a plumber or a doctor, whether or not you wish to send Xmas cards to that person, and so on. All of this information can be kept in just one field of a record, the only proviso being that no more than 16 lines of information are used, and the total length of the record is not more than 254 characters (including 'spaces').

Let us assume that you are creating an address book file such as this, using the Organiser's MAIN file – i.e., using the SAVE and FIND options from the main Menu.

When you select SAVE, the first thing the Organiser needs to know is where you wish to save your information. If you don't have a Datapak connected, you have no option. If you do have a Datapak connected, however, you have the option of saving your information in the Organiser's RAM or on a Datapak. Either way, the display is cleared and the words

```
SAVE A:
```

appear on the screen. The 'A' here is telling you that the Office is currently expecting you to save the information in the RAM memory boxes of Organiser: if you want to save onto a Datapak, you press the **MODE** key the appropriate number of times – so that the screen reads SAVE B: for the *upper* Datapak, or SAVE C: for the *lower* Datapak. If a Datapak isn't connected, the Office won't give you the corresponding option (for obvious reasons!).

Having selected where you wish to save your information, you now type it in, perhaps using the top line for the person's name, the next line for the 'phone number, the next line for the address (it doesn't matter if it takes up more than the screen's width – the information will 'scroll' to the left as it is entered), and so on, up to a maximum of sixteen lines or 254 characters.

Every time you enter a *character*, JIM rushes off with it and stores it temporarily in a safe place (called a *buffer*). If you make a mistake and go back to correct your error, JIM nips away to make the same correction in the 'safe' place. Then, when you are happy you have entered all your information correctly, the **EXE** key is pressed. This says to JIM – "O.K, that's it, go and save it properly".

JIM now goes to where all the information was kept temporarily, and transfers it into the MAIN file as one record.

Further records are added in the same way.

The situation is fairly similar when you wish to FIND one of the addresses – or any other item of information you have stored away in the MAIN file.

When you select FIND on the main Menu, the screen display switches to


```
FIND A:
```

As before when SAVE-ing information, the Office now needs to know where to look for your information – in RAM ('A'), or on a Datapak ('B' or 'C'). And just as before, the **MODE** key is used to tell it.

Having told the Office where you want it to look, it now needs a clue as to what it should look for. Let us suppose you wish to locate all the people you know with birthdays in January – and assume that you entered birthdates in the form 3/5/80 – where the month is the middle number.

With this system, *January* could be identified by '/1/' – the two '/'s being necessary to differentiate the month number from any other number that may be in the file.

So, you would type in at the keyboard so that the screen looked like this (assuming your information is in RAM and not on a Datapak):

```
FIND A:/1/
```

Pressing the **EXE** key now sends JIM off to the MAIN file on the designated 'Pack' – in this instance the RAM of Organiser II – and he meticulously examines every record to find the character sequence that you have asked for – '/1/'. As soon as he finds a match, he brings back the *entire* record – and displays it on the screen. (You won't have to wait more than a fraction of a second – JIM has eyes like a hawk). You can now use the cursor keys to examine each line of the record.

Meanwhile, JIM is waiting at the keyboard to see whether you want another record to look at, or whether you have finished. If you want another record, you'll press **EXE** again – and off JIM will go, looking through the file from where he left off, and displaying the next record that meets your requirement when finds it. When he gets to the end of the file – with no more records to look through, he will tell you:

```
**************
**END OF PACK***
```

If you wish to stop looking at some point, then pressing the **CLEAR/ON** key tells JIM you've had enough, and he puts away his

searcher's hat and gets the main Menu displayed again on the screen, ready for your next instruction.

Using the Calculator

When the CALC option is selected from the main Menu, Organiser II is set to provide facilities that emulate a powerful calculator. However, unlike an ordinary calculator, the screen display shows you what you are doing: the entry you make remains on the top line of the screen while the answer appears on the bottom line.

While you are using Organiser II as a calculator, JIM is in effect obeying your instructions as you enter them. The first thing he does is to reset the keyboard so that pressing the keys produces the 'number' characters – that is, the characters printed *above* the keys. That's because he knows that most of your entries will be numbers or mathematical symbols. Pressing the **SHIFT** key together with one of the letter keys will tell JIM that you want that letter, not the number character.

When you enter a calculation, as mentioned above, it appears on the top line of the screen – scrolling to the left if necessary to get all of your calculation on one line. The details of your calculation are stored by JIM in a safe area – the same as when entering information that you wish to SAVE. This means that you can go back and put right any mistakes you may have made in your calculation entry, using the cursor keys and the **DEL** key.

As soon as you press the **EXE** key, JIM goes to the safe place in memory where he stored your calculation, and he follows it through obeying your instructions. When he gets to an answer, he displays it on the second line of the screen.

At this point, he waits to see what you want to do next. If you press **EXE** or one of the cursor keys, he will assume that you wish to make the calculation again, but perhaps this time with a minor change (for 'what-if' calculations). So he clears the answer away from the bottom line, and places the cursor marker back on the line at the end of your calculation entry.

You can now use the cursor and **DEL** keys to make an alteration to your calculation – JIM will make the same alteration in the safe place in memory. Pressing the **EXE** key again sends him off to obey the instructions as before.

Alternatively, you may wish to hold onto that answer, for use in another calculation, perhaps. In this instance, you tell JIM by pressing the **MODE** key. Organiser II allows you ten *calculator* memory stores, and JIM needs to know which one you wish to use (you're the boss). So he asks you to make a choice, by printing a message on the top line.

```
M:  press 0-9
=your answer
```


When you've made a choice – by pressing one of the keys '0' to '9' – he then needs to know whether you want to add or subtract the current answer from anything that may already be in the chosen calculator memory, or whether you want to 'overwrite' what is there. So he presents you with another set of options:

```
M0:  +,-,EXE,DEL
=your answer
```

You now press the appropriate key (+, –, **EXE**), according to your choice. The **DEL** key clears the chosen memory (in the instance illustrated above, calculator memory '0') to zero.

You can recall the contents of the memory by simply entering **M** and the memory number while entering your calculation. Thus, to multiply the contents of calculator memory M0 by 2, you would enter:

```
CALC:M0*2
```

Note that 'calculator memory 0' does *not* mean that you are putting your information into the memory box with an address of '0'. In fact, to store a number used by the calculator takes up *eight* memory boxes altogether, and these boxes are in the RAM area of Organiser II. The ten memories that you can use with the Organiser's calculator are referred to as M0 to M9 for convenience. *They retain whatever you put in them even when you switch the Organiser off.*

When you write programs, you can put in or look into the memories M0 to M9, without having to know what their addresses are. This will be dealt with again later.

The really useful feature of Organiser's built-in calculator is that, because JIM can be sent off almost anywhere inside the Organiser, you can 'call up' any of the 'numeric functions' that are in the Organiser's programming language – and you can call up any programs you may have written yourself.

The built-in functions include practically all those you might want for engineering calculations (SINe, COSine, TANgent and so on). They also include a host of other functions which can be extremely useful.

This feature of the Calculator was used earlier on, in Chapter 1.2 under the heading 'Storing Characters', when GET was used to find the number of the pattern associated with a pressed key. The other functions you can use while Organiser is set for Calculations include HOUR, MINUTE, SECOND, DAY etc. (which give the appro-

priate current numeric value), RND (which gives a random number between 0 and 1), and, two interesting functions, ADDR( ) and FREE.

ADDR( ) needs, in the brackets, the name of a *variable* – don't worry if you don't understand what a 'variable' is at the moment. When **EXE**cuted, JIM rushes off to find the address of the *first* box holding the specified 'variable'. An example will help to explain ... how would you like to know the first address in memory where the Calculator memory 'M0' is stored? Fine, then switch on your Organiser and select CALC. Now, remembering that you must hold the **SHIFT** key down while you type in *letters*, enter ADDR(M0) and then press **EXE**. JIM will find the address for you immediately ... 8447. If you try it again, using 'M1' instead of 'M0', you'll find that the first storage address is 8455. As mentioned above, the eight memory boxes 8447 to 8454 are used to hold, in a special way, any number that you decide to keep in the calculator memory M0.

While you're in the Calculator mode, clear the current entry by pressing the **CLEAR/ON** key once, and then type FREE and press **EXE**. This instruction asks JIM to go and find out how many memory boxes are currently free for your use in the RAM area of Organiser II. It will give you an exact number of 'unused' memory boxes (or *bytes*), rather than the percentage of RAM available given by the INFO option on the main Menu. So you can keep a precise check on how much space you have left.

As you will appreciate, by providing you with access to a number of functions built into the Organiser's *programming* language, by leaving on display the calculation as you entered it, and by allowing you to go back and correct or change your calculation entry, Organiser II offers a far more sophisticated and powerful 'calculator' than any conventional calculator.

Using the other options

The way Organiser II operates to provide the other options available from the main Menu – or other built-in Menus for that matter – is similar in many respects to the general outline already given. When you have selected the required option, JIM goes off either to get another Menu, or to display the information that you have requested. If you select ALARM, for example, in order to set one of the Organiser's eight internal Alarm Clocks, he presents you with the current condition of the first Alarm (or the last one that you used). He then waits to see if you press the **EXE** key to set or reset this Alarm, or a cursor key to select another of the Alarms. If you press **EXE** on a 'free' Alarm, he will display the current day and time, for you to adjust as you wish using the cursor keys. And so it goes on.

The operation of these other options need not concern you any further: sufficient background has now been provided to give a general idea of how Organiser II operates for the built-in facilities.

1.3 Following Instructions

What is important, however, is the way Organiser II obeys your *programming* instructions, and this is dealt with in the next Chapter.

Summary

You have now been given an idea of how Organiser II provides some of the built-in options. To sum up

a) A 'Program Counter', maintained within the Office (CPU), contains the address of the memory box holding the next instruction in the Office's own language.
b) The Office work is interrupted regularly, for a few millionths of a second at a time, to enable such functions as the Clock to be updated.
c) A 'Translation Service' converts the Organiser's Programming Language – used for writing programs – into the language the Office understands (machine code).
d) The Calculator option provides direct access to a number of OPL instructions and to your own mathematical programs, as well as providing all the facilities normally expected from a calculator.

1.4

OBEYING PROGRAM INSTRUCTIONS

Programs and Procedures

We have now a classic 'chicken and egg' situation. In order to appreciate how Organiser II works when it is obeying the instructions of one of your own programs, it is first necessary to understand what a program is and how it is put together. This is given in detail in Part 3. But to understand better how to write a program, it helps considerably to know how Organiser II works when obeying program instructions – which is the subject of this Chapter.

Consequently, we shall start with a brief description of how an Organiser II program is formed.

OPL, the Organiser's own programming language, is called a *procedure-based* language. A *procedure* is simply a segment of a program – a series of instructions to undertake one particular task. A typical task could be a calculation such as miles per gallon. That task *may* be all that is required – in which case, the program consists of just one procedure.

A procedure can incorporate two or more tasks, but generally speaking, it is far better practice to make each procedure perform just one task – and to keep the procedures as short as possible. This makes writing them and testing that they work much easier.

Each procedure, when it is written, is given its own unique name. One procedure can 'call up' another procedure by simply naming that procedure (don't worry how that is done, at this stage). In other words, in obeying the instructions for one procedure, the Organiser can be 'told' to go and do another procedure: this second procedure can, in turn, call on yet a third – and so on.

That is how a complete program is built up from a number of procedures. One of the procedures may, in fact, simply 'call up' the others in the required sequence. *This* procedure would be the one that carries the name of your overall program.

You could have one task that is common to two or more completely different programs. It is not necessary to re-write the procedure for that task, with a different name, for each program that uses it. The procedure can be written just once, and 'called' by any program you choose. You can thus build up a library of commonly used procedures for use in any program that you write: one such procedure could, for example, shorten any number to two decimal places, for use in programs concerned with money matters.

Making Organiser II obey the instructions of a program (or procedure) is called *running* a program. Organiser II gives you three ways to select and *run* your program: these three ways will be discussed first, and then the manner in which the Organiser operates while running your program will be examined.

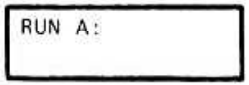
Three ways to run a program

There are three different occasions when you could want to run one of your own programs or procedures.

a) While you are programming, so that you can test that it works.
b) While you are performing a calculation.
c) As an option on the main Menu.

Organiser II allows you to run your program on any of these occasions. However, the *way* you tell the Organiser to run your program is different for each occasion.

While programming. When you are writing a program procedure, you will be in the Organiser's programming mode – having selected PROG on the main Menu. In this mode, JIM – our tireless messenger boy – presents you with another Menu, among the options of which is RUN. Selecting this option causes the screen to be cleared and the message



to be displayed. As before for the SAVE and FIND operations, JIM is in effect asking you to tell him where to look for the program procedure you want to run.

The interesting point here is that, although you must specify the location of the program procedure you wish to run, *any other procedures called by that program can be in any location.* For example, your main, controlling program could be in the RAM area of Organiser ('A') – but that program could call up procedures that are being stored on a Datapak – 'B' or 'C'. All *you* need to specify is where the main program is – Organiser sorts it out from there.

You select the location, by repeated use of the **MODE** key. JIM then waits for you to enter the name of the procedure or program you wish to run.

If you have been editing or writing a procedure immediately before you select RUN from the PROG Menu, JIM will assume that you will want to run that procedure, and will save you the effort of having to type it in by displaying its name after the RUN A: message. If he's right, all you need to do is press the **EXE** key. If he's wrong and you want to run another procedure, pressing the **CLEAR/ON** key once will clear JIM's choice so that you can type in the procedure name you want. JIM tries to be as helpful as possible.

While calculating. In this instance, you will be in the calculating mode of your Organiser, and you may wish to call up a procedure that you have written.

For example, you may wish to make a calculation using a procedure you have written that will evaluate how much an item costs *without* VAT added at 15%, given the price *with* VAT included.

You can do this by simply typing in the name of the procedure, followed by a colon: you do *not* have to specify where the procedure can be found. The colon tells JIM that he shouldn't waste his time looking for one of the Organiser's built-in functions (which are, in effect, procedures stored in the ROM memory boxes), but should instead look into the RAM area and in any Datapak that may be connected. He'll keep looking until he's searched everywhere, and if he can't find it, he'll tell you.

Usually, if a procedure or program is to be used in calculations, you would want to pass a value to the procedure for it to work on. There are a number of ways to pass values to a procedure – direct from the keyboard, through one of the calculator memories, or from the instruction that calls the procedure. All of these methods are discussed in the Programming section: sufficient to know here that JIM will obey whichever method you have selected.

Whilst in the CALC mode of Organiser II, you can also run a complete program – again, by simply typing in its name followed by a colon.

From the main Menu. One of the valuable features of Organiser II is that, having prepared your program, you can *install* it on the main Menu that appears when you switch on. In other words, it can be one of the options presented to you. You can make it the first option, too, if you wish, so that all you need to do to use your program is press the **CLEAR/ON** key, to switch on, and the **EXE** key to select and run it.

Programs used from the main Menu do not give you the option of passing information in via the instruction: if your program needs information to work on, it must arrange to get that information from the keyboard – or, less usefully, through one of the calculator memories.

When you tell JIM to **EXE**cute your program from the main Menu, he will search high and low until he finds it. Just as in the CALC mode, you don't have to tell him where to look.

After the program has ended. Generally speaking, when a program has finished JIM will reset the Organiser back to the mode it was in before the program was run. If you were in the programming mode, for example, when the program has finished he will again display the PROG Menu of options.

However, this may not always appear to be the case: your program may include an instruction to switch the Organiser off at

the end of the program – or even part way through a program. JIM will of course obey this instruction. And as always when the Organiser is off, he will wait patiently at the **CLEAR/ON** key for you to switch on again. When you do switch on, he will go back to where he left off – either part way through the program or at the end: if he was at the end of the program and Organiser II was in the programming mode, he will display the PROG Menu, not the main Menu.

Running a program

JIM obeys your program's instructions just as if they were the Organiser's own built in instructions. They will have been 'translated', remember, to the language the Office understands during the programming process.

Virtually all of your programs will need some RAM memory boxes – the ones you can change the contents of – in order to make calculations and so on. The VAT program mentioned earlier for example would be working on different information practically every time it is used. To run such a program 'in a Datapak' – even though that's where it may have been saved – would very quickly use up the memory boxes (once a Datapak memory box is 'locked up', it cannot be used again, remember). So the best place to *run* a program is in RAM.

However, it is possible that your *whole* program is built up of a number of procedures which, together, need more memory space than is available inside Organiser. Such a program would have to be kept on a Datapak.

JIM solves this problem in a clever way.

As you will see, at the beginning of every procedure that you write, you will give instructions to JIM to reserve space for the different pieces of information that your program will use. This information may be numeric values the program needs for calculations, and it may well be that you will want to enter the values from the keyboard. All the values must be saved somewhere so that the program – and JIM – can work on them.

When a program procedure is run, JIM first of all searches for that procedure – wherever it may be. When he finds it, he gets the 'space requirement' instructions and copies them into a specific area in the RAM memory boxes. He also gets the *translated* program instructions – the ones the Office understands – and copies these into RAM memory boxes too. Even if the procedure had been saved in RAM, JIM still copies it into a special area of RAM.

Then he starts to obey your program instructions, step by step. One of these instructions may name or call another procedure: if so, JIM goes away to get that procedure from wherever it is, and copies it into the special area in RAM immediately after the previous procedure – having first sorted out the space needed for information. He will then start obeying the instructions of this *second*

procedure.

When *all* of the instructions for one procedure have been completed, JIM empties the boxes those instructions occupied. They are then available to take the instructions for any other procedure that may be called.

A procedure that has had all of its instructions obeyed will *always* be the last one to have been copied into RAM: the procedure calling it will still have instructions in it for JIM to perform.

Thus even with a very long program – using a large number of separate procedures – only the procedures which JIM is actually working on are in the RAM memory at any one time. As soon as a procedure is finished with, it is cleared out of its temporary home – even if it is going to be called again umpteen times before the overall program has ended.

This technique means that the number of RAM memory boxes needed to run a program is kept to the minimum, and programs far longer than one would expect the Organiser to be capable of handling can be stored on Datapaks and run.

Don't worry if all of this seems extremely complicated to you at the moment. As you get into the programming language, it will begin to make more sense.

At this stage, it is not necessary to understand how Organiser II obeys individual instructions in the programming language: details will be given, where necessary, in the appropriate sections of Part 3.

Summary

In this Chapter, we have seen that

a) OPL, the Organiser programming language, is *procedure* based.
b) A procedure is a segment of a program (or a complete program) performing one discrete task.
c) One procedure can 'call up' another procedure.
d) A procedure, when written, can be used in more than one program.
e) Procedures or programs can be run while the Organiser is in the PROGramming or CALCulator modes, or from the main Menu if the program has been installed there.
f) When a program is run, the procedures are copied as and when they are needed, into a specific area of the Organiser's RAM, and erased from RAM once *all* the instructions in the procedure have been obeyed.

This concludes the general overview of the main components within Organiser II and the way that it operates. It is by no means a complete description, nor is it *technically* precise, but is given rather as a guide to help you better understand the programming process.

PART 2

Using the built-in applications

In this part of the book you'll find step by step instructions and tips to help you use the built-in applications provided by Organiser II's main Menu.

2.1

GETTING TO GRIPS WITH THE KEYBOARD

Not like a calculator

At first glance, Organiser II's keyboard looks rather like a calculator keyboard with an alphabet. However, it does not 'behave' quite like you'd expect a calculator keyboard to behave, and this can cause some confusion in the early stages of using your Organiser.

For example, although there is a + sign and a – sign, there are no apparent signs for multiplication or division. That does not mean, of course, that your Organiser II cannot perform these operations. It can – and to 12 figure accuracy, which is more than most calculators can achieve.

Also, you will have noticed that there is a key with = printed above it: this is not used to get the answer to a calculation, as on a calculator. Nor will the % key produce percentage calculations. You may also have noticed that there is a key labelled with a $ sign, but not a key labelled with a £ sign – that does not mean Organiser II is intended for the American market.

The reason for all these anomalies is that the Organiser II is a computer rather than a calculator, and this is reflected in the keyboard. The keyboard has been designed primarily for entering information rather than making calculations – although calculations are achieved just as easily as with a calculator.

In many respects, Organiser's keyboard is like a typewriter keyboard, the main difference (apart from layout) being that most of the Organiser's keys can produce three different characters, rather than two. The key inscribed with the letter A for example can produce **A**, **a**, or < (the symbol above the key).

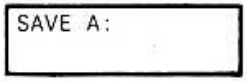
Also note that if any key is held down for more than one second, it will 'automatically repeat' very quickly indeed.

It is well worth spending a little time to become completely familiar with the keyboard. The rest of this Chapter summarises the use and purpose of various keys on the keyboard. Put them to the test for yourself – this is called 'hands on' experience, and is the best way to learn how to use your Organiser.

Selecting caps, lower case or numbers

When you first switch on your Organiser, the keyboard is set to give letters – as printed in white on the alphabetic keys.

The keys you press will not produce a display on the screen until Organiser is in a 'mode' that requires you to enter information – such as SAVE or FIND. So that you can see the effects of the **SHIFT**, **CAP** and **NUM** keys, switch on your Organiser (**CLEAR/ON** key) and press the **S** key to get into the SAVE mode. The screen should now look like this:



(If the letter 'B' or 'C' appears after SAVE, press the **MODE** key until the letter 'A' is displayed).

Immediately after the colon, you will see a flashing block: this tells you that the keyboard is set to *alphabetic* characters, and shows you where anything you type in from the keyboard will appear. As you enter characters, the flashing block – or 'cursor' as it is called – will move along to show you where characters will appear next.

Using the **SHIFT**, **CAP** and **NUM** keys to perform operations as outlined in the summary that follows, experiment typing in information from the keyboard. Do *not* press the **EXE** key – else you will SAVE the information that you type in!

To clear any entry that you have made, press the **CLEAR/ON** key once: pressing this key twice will return you to the main Menu.

Note that, if you enter more characters on one line than the line length, the display will 'scroll' to the left. You can use the cursor keys (discussed in the next section) to move about your entry or to the next line.

The **SHIFT**, **NUM** and **CAP** keys function as follows:

SHIFT and NUM keys pressed simultaneously

This combination sets the *whole* keyboard either to aphabetic characters (those printed in white on the keys) or to numeric characters (those printed above all alphabetic keys), depending on the previous setting.

You would use this combination before making a large number of numeric or alphabetic entries.

SHIFT and CAP keys pressed simultaneously

This combination switches *alphabetic* inputs from the keyboard between capital letters and lower-case (small) letters.

You would use this combination whenever you wished to change from capitals to lower case or vice versa.

SHIFT plus any 'white' key

Using the **SHIFT** key simultaneously with any of the keys carrying a white letter produces the *alternative* character to that for which the keyboard has been set.

Example 1: Keyboard set to alphabetic inputs.

Pressing **K** produces 'K' or 'k', depending on whether the keyboard is set for capitals or lower case.

Pressing **K** and **SHIFT** produces '9', the character above the 'K' key.

Example 2: Keyboard set to numeric inputs.
Pressing **K** produces '9'.
Pressing **K** and **SHIFT** produces 'K' or 'k', depending on whether the keyboard is set for capitals or lower case.

Note: For all Menu displays, the keyboard is automatically set for alphabetic inputs (capitals or lower case, depending on previous use of the **SHIFT** and **CAP** keys).
For the Calculator, the keyboard is automatically set for numeric inputs.

Tip: *When entering mainly one type of character, set the keyboard for that character type (alphabetic or numeric), and use the* **SHIFT** *key to produce the alternative characters when needed.*

The cursor keys

The keys marked with arrows – pointing up, down, left and right – are used to control the position of the cursor on the screen. The cursor indicates where the next character to be typed will appear on the screen. The use of the cursor keys is fairly obvious: the **RIGHT** and **LEFT** arrow keys move the cursor one position to the right or left respectively. The **UP** and **DOWN** arrow keys move the cursor to the *beginning* of the line above or below its current position.

Sometimes you will find that the cursor will not move past a certain point: this means that there is no information available past that point, or that it is not possible to enter information past that point.

When used with a Menu display, the **RIGHT** and **LEFT** cursor keys move the cursor one word at a time, so that the cursor is always over the first letter of an option.

The **RIGHT** and **LEFT** cursor keys can also be used to control a line on the screen when it is scrolling – that is 'rolling' round so that you can see the entire entry should it be longer than 16 characters. Using the appropriate right or left key will stop or start the scroll in the corresponding direction.

The CLEAR/ON key

Apart from its obvious use of switching Organiser II on, for the built in applications this key generally has the effect of *clearing* Organiser back to its previous state. For example, during SAVE and FIND operations, if information has been entered from the keyboard, pressing **CLEAR/ON** once will clear that information from the display. Pressing **CLEAR/ON** the second time returns to the main Menu.

There are two exceptions:

When editing a program
The **CLEAR/ON** key removes the current line only: that is, the line on which the cursor is displayed.

When using the Diary
When the Diary is initially selected, the screen displays the current date and time (to the next half hour segment).
From this condition, **CLEAR/ON** returns to the main Menu.
If editing a Diary entry:
The first press of **CLEAR/ON** clears that entry.
The second press clears the EDIT operation.
The third press returns to the main Menu.

The MODE key

Generally speaking, this key provides selection of the non-Menu options available within an application. Its main uses are as follows:

Main Menu
Pressing **MODE** allows you to add or replace an option on the main Menu – if you add an option, there must of course be a program for it in the Organiser.

Applications involving a 'location'
The **MODE** key allows selection of the desired location ('A' for the internal RAM, 'B' or 'C' for Datapaks fitted to the upper and lower slots respectively). If Datapaks are not plugged into Organiser II, the selection obviously cannot be made.

Diary applications
When the screen is displaying a date and time – pressing the **MODE** key will switch the display to the DIARY Menu.

Alarm applications
Pressing **MODE** whilst setting one of the eight Alarms switches on the 'repeat' action for the Alarm – that is, enables the Alarm to be set to ring hourly, daily or weekly.

Time application
Pressing **MODE** when TIME has been selected enables the time and date to be set or reset.

The DELete key

This key is used to delete entries from the Organiser (you guessed!). Its use is summarised below:

On the main Menu
Pressing **DEL** removes the option under the cursor from the displayed Menu. The option can be restored (by using the

MODE key), but while it is deleted, the option cannot be used.

While editing

When editing an entry, this key is used to delete the character to the left of the cursor.

SHIFT and **DEL** when pressed simultaneously delete the character under the cursor.

Diary applications

Apart from its use to delete characters whilst editing, the **DEL** key can be used to delete a previously recorded entry.

Erasing records

In order to delete a record that has previously been SAVEd, it is necessary to first select the ERASE option from the main Menu, then select the record to be deleted (by repeated use of the **EXE** key or by specifying a search clue as for FIND), before pressing the **DEL** key.

Since deleting a record can be a drastic action if undertaken by mistake, Organiser will seek confirmation that you want the record deleted by asking DELETE Y/N. Pressing **Y** makes the deletion, **N** cancels it.

The EXEcute key

This is the 'all-action' key. It tells Organiser II to go ahead and perform the operation selected.

For example, after typing in information during a SAVE operation, pressing **EXE** will cause that entry to be stored in RAM. After entering a 'clue' during a FIND operation, pressing **EXE** will tell Organiser II to go and search for the required information. After entering a calculation when Organiser II is set to CALC, pressing **EXE** will cause the calculation to be made.

Further explanation is unnecessary.

Special keys

A number of the 'numeric' keys have special functions when used for the CALCulator or PROGramming applications.

The '*' key, for example, is used for multiplication (not the 'x' key, as you might imagine), and the '/' key is used for division.

An explanation of the use of these keys is given in the appropriate sections of this book.

2.2

THE MAIN MENU

Switching on

When you first switch on Organiser II (using the **CLEAR/ON** key), the screen will display a series of options – covering the applications immediately available to you:

```
FIND SAVE DIARY
CALC PROG ERASE
```

These are not the only options available. By using the **UP** and **DOWN** cursor keys, the screen display will scroll to reveal further options available. The full list of options is

```
FIND SAVE DIARY
CALC PROG ERASE
ALARM TIME INFO
COPY RESET OFF
```

TIME gives you a display of the current date and time, while ALARM allows you to set or reset up to eight different alarms, without repeats or with repeat periods of every hour, day or week. These two applications are dealt with in Chapter 2.3.

The SAVE, FIND, ERASE and COPY options enable you to record, find, change and manipulate information you wish to keep – such as an address list with telephone numbers, birthdays and so on, or perhaps gardening information or recipes. This application is covered in Chapter 2.4.

DIARY provides you with half-hour segments through to the year 2000, so that you can record appointments and so on. It also gives you facilities for editing your entries, browsing through them, finding a specific entry on the flimsiest of information and, perhaps most useful of all, it allows you to set an alarm call up to 59 minutes before all or any of your appointments. This alarm facility is quite independent of the eight alarms provide by the ALARM option. The DIARY is discussed in Chapter 2.5.

CALC provides you with a powerful calculator which, unlike other calculators, shows you what you are doing and allows you to 'edit' your calculation so that you can repeat it with different figures. It gives you access to a range of scientific and mathematical functions, and has ten of its own memories (the contents of

which are kept even when you switch off). It also lets you use your own programs to perform calculations – with prompts, if you wish. The CALCulator is covered in Chapter 2.6.

PROG provides the extremely powerful programming facility, allowing you to tailor the Organiser II to do the things you specifically want. This is dealt with in Part 3 of this book.

Finally, there are the INFO, RESET and OFF options, which need only a brief explanation, and are dealt with in this Chapter.

Selecting an option

Two methods are available for selecting the facility you wish to use.

Method 1

Use the cursor keys to position the cursor (flashing blob) over the first letter of the required option, then press **EXE**.

Method 2

Type in the first letter of the option you wish to use.

If only one option begins with that letter, the option will be selected immediately.

If more than one option begins with the entered letter, the cursor will move to the first of such options. In this instance, press the letter key until the cursor covers the required option, then press **EXE**.

Note: If using method 2, the cursor must be a flashing blob: if for any reason the cursor is just an underline (indicating 'numeric' inputs), press the **SHIFT** and **NUM** keys simultaneously to restore the keyboard for alphabetic inputs. It does not matter if the keyboard is set for capital or small letters.

Customizing the main Menu

It may well be that the order of the options on the Menu is not to your liking. For example, you may prefer to have the DIARY as the first option so that you can access it straight away by simply switching on and pressing **EXE**.

Also, you may want to remove options from the Menu (the RESET option can erase all the information you wish to keep, if you are not careful. Removing it from the Menu eliminates the possibility altogether). And when you have written a program, you would no doubt like to add that program as an option on the Menu.

You can customize the main Menu exactly how you want.

Note that, when an option is removed from the Menu, it cannot be accessed by Methods 1 or 2 given previously. It is not *permanently* lost, however: to regain that option, you need only to re-install it on the Menu.

Deleting an option

a) Use the cursor keys to position the cursor over the option you wish to delete.
b) Press the **DEL** key
c) The screen will display the option on the top line, and DELETE Y/N on the second line – asking for confirmation that you do indeed wish to delete the option (you may have pressed **DEL** in error).
d) To make the deletion, press **Y** (*Yes*). To cancel the operation, press **N** (*No*).

Note: The OFF option cannot be deleted, for obvious reasons.

Restoring (or adding) an option

a) Use the cursor keys to position the cursor at the place in the Menu where you wish your option to be restored (or added).
b) Press the **MODE** key.
c) The screen will display INSERT ITEM on the top line, and the cursor will be positioned at the beginning of the second line.
d) Carefully type in the name of the option you wish to restore or add, and press **EXE**: it doesn't matter whether you use capital or lower case letters – Organiser will change them to capital letters when displaying the Menu.
e) The option will now be included in the Menu: other options will have been 'moved along' to make room for the addition.

Note: If you add an option name which is not one of the Organiser's built-in applications, or is not one of your own programs, when you try to select this option the screen will display the message:

```
MISSING PROC
  'option'
```

'PROC' is short for procedure, and Organiser II is telling you that it cannot find the procedure or program that you have selected anywhere. If it is simply a question of incorrect spelling when you added the option, delete it and re-install it correctly in the Menu. If the program doesn't exist, there is no point in trying to include it.

Note too that you cannot delete one of the Organiser's options and replace it with one of your own programs having the same name: Organiser II will assume you want to install its own built-in option.

Moving options around

a) Follow the *Deleting an option* instructions to remove the option from its current position.
b) Follow the *Restoring (or adding) an option* instructions to re-install the option in the desired position.

Tip: When adding one of your own programs as an option, try to keep the initial letter different from any already existing in the Menu: this will make selection easier when using the 'initial letter' method. Note that you cannot rename any of the built-in options.

The INFO option

When selected, this option tells you on the top line of the screen the total number of RAM memory boxes available in Organiser II for *your* use. It will be about 1000 less than the actual number built in, because Organiser itself needs some RAM space in which to work.

The second line of the screen gives you a guide as to how much of the RAM space is occupied by the Diary and any records or programs you may have stored, and also, if fitted, how much space you have used in the Datapaks. The display scrolls continuously to the left, and shows you the percentage of space *used*, ending with the percentage of space free in Organiser's RAM. Use the **LEFT** and **RIGHT** cursor keys to stop the second line from scrolling, or to start it again in the desired direction.

To find the exact number of *bytes* (memory boxes) you have free in Organiser's RAM, select the CALC option from the main Menu, and remembering to keep the **SHIFT** key pressed to enter letters (CALC automatically sets the keyboard for numeric inputs), type in FREE and press **EXE**.

To return to the main Menu, press **CLEAR/ON**.

The RESET option

There may be an occasion when you want to completely clear out everything that you have stored in the RAM of your Organiser – so that it is just as it was when you switched it on for the very first time.

The option for this is RESET.

However, RESET can be a drastic measure if used in error. Consequently, Organiser II makes doubly sure you really do want to reset and start again.

When RESET is selected, the screen will display the message:

```
ALL DATA WILL BE
LOST - PRESS DEL
```

This is the first reminder that you are about to lose all your information.

If you do not wish to continue, press **CLEAR/ON** – you will be returned to the main Menu and nothing will be lost.

To continue, press **DEL**. The screen will now display


```
ARE YOU SURE
PRESS Y/N
```

This is your last chance to back out. Pressing **N** – for No – will return you to the main Menu. Pressing **Y** will erase all the information stored in Organiser's RAM, including the Diary, any Alarms you may have set, any records you may have stored, any programs you may have written, and anything you may have stored in one of the calculator's ten memories. None of this information is recoverable: it will have to be re-entered.

Information stored on any Datapak will *not* be affected. Nor will the internal clock – you will still have the correct time available.

Tip: To be sure you won't select the RESET *option in error, remove it from the main Menu: you can always re-install it again should you wish to use it.*

Switching off

This is as simple as pressing the **O** key from the main Menu. Alternatively, you can use the cursor keys to select OFF on the main Menu, then press **EXE**.

If Organiser II is left on for five minutes without any keys being pressed, it will decide you've been interrupted in whatever you were doing, and it will switch off on your behalf – to conserve battery power.

All is not lost. When you switch on again – using the **CLEAR/ON** key as normal, you will be returned to exactly where you left off, as if nothing had happened.

2.3

TIME AND ALARMS

Keeping TIME

The date and time in Organiser II is continuously updated, even when Organiser is switched off. The date and time information is used by the Alarm application, and by the Diary to provide the current half hour segment when this application is selected. Date and time information is also accessible from Organiser's built in programming language, so that it can be used in your own programs if required.

To set the date and time

a) Select the TIME option from the main Menu.
b) Press the **MODE** key: the cursor will appear over the day of the month, and the clock will be *stopped*.
c) Use the **UP** and **DOWN** cursor keys to set the correct day of the month: the **UP** cursor key increases the figure one step at a time, the **DOWN** cursor key decreases the figure one step at a time.
Note that the name of the day in the week changes automatically as the date is changed.
d) Use the **RIGHT** (and **LEFT**, if necessary) cursor key to position the cursor over the name of the month.
e) Set to the current month by using the **UP** and **DOWN** cursor keys.
f) Continue as in d) and e) above, using the **RIGHT** and/or **LEFT** cursor keys to position the cursor over the item to be changed, and the **UP** and/or **DOWN** cursor keys to adjust the setting, until the date, hours and minutes have been set.

Note: Organiser II uses the 24 hour clock, so that 1.27.00 pm, for example, should be set as 13.27.00.

g) When the date and time are set correctly, press **EXE**: the clock will be re-started instantly. The clock can thus be set precisely by pressing **EXE** when the actual time (as obtained from the 'Speaking Clock', for example) matches that registered on the Organiser.
h) Press **CLEAR/ON** to return to the main Menu.

The correct date time will now always be available by selecting the TIME option from the main Menu.

When changing batteries, the date and time within Organiser II will continue to be updated for approximately 30 seconds, giving

you time to make the change without need to reset the internal clock again afterwards. Alternatively, the optional power supply lead can be connected whilst the battery change is made, so that the clock is kept running however long the change takes.

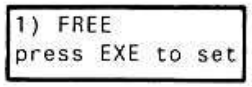
Using the Alarms

There are eight individual Alarms in Organiser II. All or any of them can be set as described here. Note that these Alarms are quite independent of the Alarms that can be set in the Diary application, and give a different 'buzz'.

Any Alarm can be set for up to one week ahead, and can be set to repeat every hour at the same number of minutes past the hour, every day at the same time, or every week on the same day and same time.

Setting an Alarm

a) Select the ALARM option from the main Menu.
b) The screen will display



If Alarm 1) has been set previously, the screen will display the day of the week, hour and minute of the setting on the top line, and any 'repeat' instruction on the bottom line.
c) Use the **UP** and **DOWN** cursor keys to select the Alarm number to be set.
d) Press **EXE**.
e) The current day of the week and time will be displayed on the top line, with the cursor flashing over the day of the week.
f) To set the Alarm, use the **LEFT** and **RIGHT** cursor keys to select the item to change, and use the **UP** and **DOWN** cursor keys to effect the change.
To cancel the Alarm *before* it has been set, simply press **CLEAR/ON**.

For a 'once only' Alarm

Press **EXE**. The cursor will be removed from the screen. Now,
1) To set another Alarm, repeat these instructions from c).
2) To return to the main Menu, press **CLEAR/ON**.

For repeated Alarms

Press **MODE**. The letter **R** will appear under the day of the week. The **RIGHT** and **LEFT** cursor keys can be used to position the **R** under the hours or minutes, to obtain repeated Alarms as follows.

Once a week, on set day and time: position the **R** under the day name.
Every day, at set time: position the **R** under the hours figure.
Every hour, at set minutes past the hour: position the **R** under the minutes figure.

To cancel the repeat function before it has been set, press **MODE** again.

To enter the Alarm and repeat function into Organiser's memory, press **EXE**. Then:

1) To set another Alarm, repeat these instructions from c).
2) To return to the main Menu, press **CLEAR/ON**.

Cancelling an Alarm

a) Select ALARM from the main Menu.
b) Use the **UP** and **DOWN** cursor keys to locate the Alarm to be cancelled.
c) Press **DEL**.
d) Use the **UP** and **DOWN** cursor keys to locate another Alarm for cancelling (or setting), or **CLEAR/ON** to return to the main Menu.

Modifying an Alarm

a) Select ALARM from the main Menu.
b) Use the **UP** and **DOWN** cursor keys to locate the Alarm to be modified. You then have a choice:

To change the day or time of alarm

1) Press **EXE**.
2) Use the **RIGHT** and **LEFT** cursor keys to select the item to change, and the **UP** and **DOWN** cursor keys to effect the change.
   If the repeat function has been set on the selected Alarm, the **R** indicator will move with the operation of the **LEFT** and **RIGHT** cursor keys. Once the Alarm has been reset as required, simply use the **LEFT** and **RIGHT** cursor keys to restore the position of the **R** indicator.
3) Press **EXE**, then **UP** and **DOWN** cursor keys to select another Alarm, or **CLEAR/ON** to return to the main Menu.

To cancel/insert the repeat function

1) Press **MODE**. If cancelling, the **R** will disappear and the Alarm will be set 'once' only. If adding, the **R** will appear, and can be positioned as detailed previously, using the **LEFT** and **RIGHT** cursor keys.
2) Press **EXE**, then the **UP** and **DOWN** cursor keys to select another Alarm, or **CLEAR/ON** to return to the main Menu.

2.4

KEEPING RECORDS

The built-in filing system

One of the important features of Organiser II is its ability to store information, and to enable stored information to be found with only a small 'clue' as a guide.

Organiser provides complete flexibility for you to design your own 'storage' system, using the programming language. It also provides a built-in system which enables an extremely useful 'file' of information to be created. This Chapter deals with the built-in system.

A 'file' can be imagined as an empty card index box. A 'record' is a card of information placed in the index box. When you SAVE information in your Organiser II, you are effectively writing out a record card and placing it in a card index box.

When a power supply is first connected to the Organiser, an empty 'file' is created in its RAM memory. When Datapaks are first connected, empty 'files' are created on these too. Connecting two Datapaks effectively gives you three 'card index box' files – one in RAM, and one on each of the Datapaks.

As a matter of interest, all these files have the same name – MAIN – but you don't have to worry about that until you write programs to create your own files. What is important, if you have Datapaks connected, is that each file or 'card index box' is identified by its *location*. The file in RAM is always identified by 'A:', that on the upper Datapak by 'B:' and that on the lower Datapak by 'C:'. Note that it is the *location* of the Datapak that is important – not what it contains.

Thus if you have an address list in the MAIN file on a Datapak in the upper slot, you would identify the file for that Datapak as 'B:'. If the same Datapak were removed from the upper slot and plugged into the lower slot, the file on that Datapak would be identified by 'C:'.

If Datapaks are connected, you choose which 'file' your information goes into. If Datapaks are not connected, naturally you have no choice.

The main Menu options which allow you to use your files are SAVE, FIND, ERASE and COPY.

SAVE is the option to choose when you wish to store away information – in effect, to write out a record card and stow it in the index box.

FIND enables you to locate a specific record quickly by giving the smallest of clues as to what you want, and, if you wish, lets you change that record.

ERASE lets you remove completely one or more records from your file.

COPY lets you copy information from one file to another – from the MAIN file in RAM to the MAIN file on a Datapak, for example.

Storing your information

If you have Datapaks connected to your Organiser, you should think very carefully about the type of information you are going to store on them. The information stored in Organiser's RAM can be changed, modified or even deleted without losing memory space. However, whenever a change is made to information held in a Datapak, the entire original record is 'locked up', and a new record created.

Taking the card index box analogy, in the RAM card index box, information on each individual record card can be 'rubbed out' and replaced by new information, or the card removed completely. In a Datapak card index box, cards are left in the box, and the new information is 'written' to a new card: the original, now out-of-date card stays in the box taking up space, but cannot be examined or looked at any more. Thus a Datapak can gradually get filled up with obsolete information.

Hence, if you wish to keep information which changes quite frequently, you should save it in Organiser's RAM ('A:'). The Datapaks are best used for information of a more permanent nature.

In any event, it is advisable always to save any information *initially* in RAM, and to transfer or copy it to the desired Datapak when you are sure it is correct.

When saving information, think ahead to how you may wish to use that information. For example, let us say you wish to store names, addresses and telephone numbers – not just of family and friends and/or business associates, but also important local people – such as the Doctor, plumber and so on. You may also wish to keep on record the birthdays of those close to you, and perhaps have easy reference of all the people you wish to send Christmas Cards.

Provided the required information is stored in a consistent manner in your Organiser, it can be found again very easily.

Thus if you know you will want to find, for example, all the people with birthdays in the current month, make sure you enter *all* birth dates the same way – 12/3/1940 or 12 MARCH 1940, whichever

you prefer. Use some form of simple code to identify groups of people you may wish to find: for example, you can identify those you wish to send a Christmas card to by 'XC, or perhaps just an asterisk '*'.

Similarly, if you save the name and telephone number of local tradesmen, identify them by including their trade or profession ('Plumber', 'Decorator', 'Doctor') in the information so that you will still be able to find them quickly even if you cannot remember their name. And so on.

Remember too that Organiser will display the top two lines of information when it finds a record. Although you can quickly access the rest of the information, it is best to put the most important details on these two lines so that you can see them straight away.

Also, it is advisable to restrict one record to one 'set' of information – one person's name, address and so on.

These little 'tricks' will make it extremely easy to locate information you may need in a hurry, or to get Organiser to list for you all the people of a certain category (with birthdays in MAY for example).

To SAVE a record

a) Select SAVE from the main Menu.
The screen should display the message SAVE A:. This tells you your information will be saved to the 'A' Pack – Organiser's RAM. To save to a Datapak, press the **MODE** key until the required Datapak is selected ('B' for a Datapak in the upper slot, 'C' for a Datapak in the lower slot). But note that it is not usually advisable to use Datapaks to save information that can change frequently.

b) Type in your top line of information. If you are entering an address, this would be the person's name, possibly followed by their profession (i.e. DOCTOR). Notice that the screen display will 'scroll' to allow you to enter more information than the normal screen width.

c) To enter information on the next line down, use the **DOWN** cursor key to position the cursor on that line. You may choose to use this line for the telephone number. (See Chapter 2.1 regarding the use of the **SHIFT**, **CAP** and **NUM** keys to enter capital or lower case letters, or numbers).

d) Continue entering information until your entire record is complete. Remember to add any special 'codes' you may wish to use to locate that person – for example, '*' as a recipient of Christmas cards.

e) Use the cursor keys to move round your record and to edit it (in conjunction with the **DEL** key).

f) To delete a record – without saving it – press **CLEAR/ON**: you will still be in SAVE, and can re-enter your record if you wish. To return to the main Menu, press **CLEAR/ON** again.

g) When you are sure the information is correct, press the **EXE** key. This enters your record into the selected file.

Note: Records can be up to a maximum of sixteen seperate lines or a maximum of 254 characters. You could, if you wished, have the entire record on one line – but this would make the record less easy to read when you are examining it later.

h) To enter another record, simply repeat this procedure from the start.
Note that you do *not* have to enter your records in alphabetic – or any other – order.

Locating information

The real power of Organiser II becomes apparent when you wish to locate one or more of your records – on a specific item of information.

The methods for locating records are as follows:

To FIND a specific record

a) Select FIND on the main Menu.
The screen will display FIND A:. If the information you want to find is not in the Organiser's RAM ('A'), select the required Datapak by pressing the **MODE** key. (If Datapaks are not connected, this will have no effect).

b) Type in just a few letters or symbols that you know are included in the record you wish to find. For example, to find the Doctor – simply enter DOC. To find those people you wish to send Christmas cards to, enter your 'code' for the Christmas card recipients. To find a person by name, enter just a part of that name.

c) Press **EXE**. Organiser will immediately search through the selected file for a match, and will display the first matching record it finds on the screen.

d) If this is not the record you want, or if you wish to find another record with the same match, press **EXE** again, repeating as necessary. If a match is not found, or the end of your records is reached, an END OF PACK message will be displayed on the screen.

e) Press **CLEAR/ON** to return to the main Menu.
Press **EXE** to continue the search from the start again.

To browse through your records

a) Select FIND on the main Menu.
The screen will display FIND A:. If the file you wish to browse through is on a Datapak, use the **MODE** key to choose the appropriate Datapak.

b) Press **EXE**. The first two lines of a record (usually the first record) in the selected file will be displayed. (See the the next

section – *When a record has been found* – for details of how to examine your record).

c) Each successive press of **EXE** will display a new record, until you reach the END OF PACK message. Pressing **EXE** again will cause the first record to be displayed (depending on previous use of Organiser, FIND may *not* always start by locating the first record in your file).

When a record has been found

a) If the top line of your record is more than 16 characters long, that line will 'scroll' to the left, so that you can read it all. Use the **LEFT** and **RIGHT** cursor keys to stop and start the scrolling action.

b) Use the **UP** and **DOWN** cursor keys to view the other lines of your record: if any line that the cursor is on is longer than 16 characters, that line will scroll. Use the **LEFT** and **RIGHT** cursor keys to stop and start the scrolling action.

c) To find the next record, press **EXE**. To return to the main Menu, press **CLEAR/ON**.

Changing the information in a record

You can change the information contained in any of your records at any time. However, note that if the record is kept in a Datapak file, the old record will not be 'erased' from the Datapak, but will be 'locked up' so that it can no longer be examined. It will still occupy space in the Datapak.

The procedure for changing information in a record is as follows.

Editing a record

a) Select FIND on the main Menu.
The screen will display FIND A:. If the record you wish to change is on a Datapak, use the **MODE** key to select the appropriate Datapak.

b) Enter a 'search clue' to locate the record you wish to amend. For example if the record you wish to change contains the name 'Bloggs', you could type in 'oggs' or 'Blog'.

c) Press **EXE**. The first two lines of the required record should be displayed. If an incorrect record is displayed (because it, too, contains the 'search clue' that you entered), press **EXE** until the required record is found.

d) Press **MODE**.
The screen display will now change: the first line of your record will start with the message SAVE A: (or SAVE B: or SAVE C:, depending on whether a Datapak has been selected).

e) Use the cursor keys to locate the information you wish to change.

f) Use the **DEL** key, or the **SHIFT** and **DEL** key to remove the information you wish to change.

g) Enter the replacement information in the appropriate place, and check that it is correct.

Note: At any time from step d) to this point, you can abort making a change by pressing **CLEAR/ON** *twice*. Alternatively you can delete the *entire* record by pressing **CLEAR/ON** once, followed by pressing **EXE**.

h) If the amended record is to be saved back to the same place it came from – i.e. Organiser's RAM, ('A:') or a Datapak ('B:' or 'C:') – press **EXE**.
If the amended record is to be saved to the MAIN file at an alternative location (i.e. to a fitted Datapak), press **MODE** to select that location and then press **EXE**.

Removing records from a file

Organiser II allows you to locate and delete a single record, to locate and delete a group of records with a common entry, or to browse through your file and delete records as you wish.

The procedure for deleting a single record or a group of records with a common entry is the same:

Deleting one record/common entry records

a) Select ERASE on the main Menu.
The screen will display ERASE A:. If the file containing the record(s) you wish to delete is on a Datapak, use the **MODE** key to select the location of the Datapak.

b) Type in three or four letters as a 'search clue' for the record(s) you wish to delete. Thus to delete a record or all records containing the word 'Temporary', you could type in 'Temp' or 'mpor'.

c) Press **EXE**.
If the record displayed is not one you wish to delete, press **EXE** repeatedly until the correct record is found. (Other records may include the same 'search clue' you entered).

d) To delete the record, press **DEL**.
The second line of the screen display will now be replaced by DELETE Y/N – checking that you do indeed wish to delete this record.

To confirm the deletion, press **Y**.
The record will be deleted from the file, and either another record with the same match will be found and displayed, or the END OF PACK message will be displayed.
To delete the next record found, repeat from d) above.
To return to the main Menu, press **CLEAR/ON**.

To avoid deleting the record, press **N**.
The record will not be deleted from the file, but will still be displayed on the screen.
To continue the search, press **EXE** – i.e., continue from step c) above.

e) To abort the search, press **CLEAR/ON**. You will be returned to the main Menu.

Browsing through to delete records

This procedure is identical to the one given previously, except that you press **EXE** without entering a search clue when ERASE A: is displayed on the screen. You will be presented with all your records – a new one each time you press **EXE**.

To make a deletion, press **DEL**, as in the previous procedure.
When a deletion is made, the next record will be displayed.
To finish browsing at any point, press **CLEAR/ON**.

Copying a complete file

When Datapaks are connected, you may wish to transfer an entire file of records from one location to another.

The COPY option on the main Menu allows you to do this. (You can copy *any* file using the COPY option, but the file must first have been *created* – through a program that you write).

The procedure is as follows:

a) Select COPY on the main Menu.
The top line of the screen will display FROM.

b) You must first type in the *location* of the file you wish to *copy*.
If the file is in Organiser's RAM, type in **A:**. If the file is in a Datapak, type in **B:** or **C:**, depending on whether the Datapak is connected to the upper or lower slot respectively.

c) If you wish to transfer all *files*, or if you have only *one* file at the specified location – (as will be the case until you create your own files), press **EXE**.
To specify a particular file that you wish to transfer, type in the name of the file. (You could do this with the built-in file by entering the built-in file's name – MAIN). Then press **EXE**.

d) The word TO will be displayed on the second line of the screen. You must now enter the location you want the file *copied to* – **A:**, **B:** or **C:**.
This location must be different from the *source* location.

e) If the copied file is to have the same name, press **EXE**. If you wish to give the copied file a new name, type in the new name, then press **EXE**.

Note: If you rename the MAIN file, the SAVE, FIND and ERASE options on the main Menu will *not* be able to access the copied file. The renaming facility is provided for the files that you create yourself when programming Organiser II.

f) The screen will clear and the message 'Copying...' will be displayed. The copying process may take some time, depending on the number of records in the file to be copied.

Note: Copying a file places a higher drain on the battery power: if the **LOW BATTERY** message is displayed, press **CLEAR/ON** to return to the main Menu, and switch off (press **O**) *immediately*. Failure to do this could result in there being insufficient back-up power to retain your records, your Diary information and the time and date.

g) If the file already exists at the destination, the copied records will be *added* to the end of that file. If the file does not exist at the destination, a file will be created automatically either with the specified name, or if a name hasn't been specified, with the same name as the file being copied.

Here are some examples (You don't enter 'FROM' or 'TO').

1)

```
FROM A:
TO B:
```

Every file in Organiser's RAM is copied to the Datapak fitted in the upper slot. Any records in the MAIN file in RAM are *added* to any records in the MAIN file on the Datapak.(MAIN files are automatically created on Datapaks when they are first plugged in).

2)

```
FROM A:MAIN
TO C:
```

Only the MAIN file in Organiser's RAM is copied to the MAIN file in the Datapak fitted to the lower slot. As for example 1), records in the RAM file are *added* to those in the Datapak.

3)

```
FROM B:MAIN
TO A:OTHER
```

The records in the MAIN file on the upper Datapak are copied to a file called 'OTHER' in Organiser's RAM. If a file called 'OTHER' already exists, the records will be

added to it. If the file doesn't exist, it will be created, and then the records copied into it. The built-in SAVE, FIND and ERASE options will *not* access the file called 'OTHER': it can be accessed only from your own programs.

Copying one MAIN file record

You may wish to copy just one of your SAVEd records to or from a Datapak. For example if a 'changeable' record kept in RAM becomes more permanent, you may wish to transfer it to a Datapak. This can be achieved using the FIND option on the main Menu, as follows.

a) Select FIND on the main Menu.
The screen will display FIND A:. Use the **MODE** key to select the location of the record you wish to copy – 'A:' if it's in the RAM file, 'B:' or 'C:' for the Datapaks.

b) Type in three or four letters as a search clue for the record. Thus if you wish to copy a record for a person called 'Jones', you could type in 'Jon'.

c) Press **EXE**.
The first two lines of the record you wish to copy should now be displayed: if the wrong record is displayed (because it, too, contained your search clue), press **EXE** repeatedly until the required record is displayed.

d) Press the **MODE** key.
The first line of the display will now start with the word SAVE, followed by the current location of the record – 'A:' for RAM, 'B:' or 'C:' for the upper or lower Datapak respectively.

e) Press the **MODE** key until the required destination location is displayed. Thus if you wish to save the record to a Datapak fitted into the lower slot, press **MODE** until the first line of the display starts with SAVE C:

f) Press **EXE**.
The displayed record will now be added to the MAIN file at the specified location.

Note: If the record is copied back to its original location, or if it already exists at the new location, it will be in the MAIN file *twice*. In terms of the card index box, you will have written out another record card duplicating the first, and placed that in the box too. This is obviously a waste of space – and is *not* the way to correct a record.

g) After the record has been copied, you will be returned to the main Menu.

2.5

KEEPING A DIARY

The Diary Menu

The Diary enables entries to be made right up to the last minutes of the year 1999.

Entries are made in half-hour 'slots': in other words, the Diary can be considered as an appointment book with space for making an entry every half-hour throughout the day.

When an entry is made in the Diary, you have the option of requesting an Alarm to remind you of the entry. This Alarm can be set to buzz from one to 59 minutes before the entry time, and can be used for as many – or as few – entries as you wish. The Diary alarms are independent of and produce a different sound to the eight Alarm clocks built into Organiser II.

When the DIARY option is selected from the main Menu, the screen displays immediately the current date and time slot on the top line: if any entry has been made within this slot, it is displayed on the lower line.

This display is in the PAGE mode of the DIARY Menu, described later.

To obtain the DIARY Menu, press **MODE**.

(To return to the main Menu, press **CLEAR/ON**.)

The options offered on the DIARY Menu cannot be changed, removed or added to, as is the case with the options on the main Menu.

The options offered are:

```
PAGE LIST FIND
GOTO SAVE TIDY
RESTORE DIR
ERASE
```

The options can be selected in the same way as those on the main Menu: that is, either by entering the first letter of the required option (the fastest and easiest way), or by using the cursor keys to place the cursor over the first letter of the required option, then pressing **EXE**.

Briefly, the function of each option is as follows:

PAGE	Enables you to make your Diary entries.
LIST	Enables you to browse through your Diary entries, starting from the *latest setting* of the date and time.
FIND	Enables you to find a specific Diary entry or entries, from a three or four letter search clue.

GOTO	Enables you to select any specific *date* within Organiser's range (1 January 1900 to 31 December 1999).
TIDY	Enables you to erase from your Diary all entries up to the *latest setting* of the time and date.
SAVE	Enables you save a *copy* of the entire contents of your current Diary to RAM or a Datapak.
RESTORE	Enables Diary information that has been SAVEd to be put back into RAM as the *current* Diary – *overwriting* the existing current Diary information.
DIR	Enables you to see a list of all the Diary names you may have SAVEd.
ERASE	Enables you to delete a Diary that you have SAVEd.

The first five of these options enable you to work on the *current* Diary – that is, the one that Organiser II looks after on your behalf. Each of these options is discussed separately, followed by the procedures for using your Diary.

The last four options enable you to keep more than one Diary: however, it should be noted that Organiser II will display information and respond to Alarm calls in the *current* Diary only. These options are dealt with together in the section *Keeping more than one Diary*.

Key operation from the Diary Menu

The keys function as follows whilst the DIARY Menu is displayed on the screen.

CLEAR/ON	Selects the PAGE mode of the Diary.
Cursor keys	Enable an option to be selected.
EXE	Selects the option indicated by the flashing cursor.
Letter keys	If the pressed letter key corresponds to the first letter of an option, that option is selected.

Other keys have no effect.

The PAGE option

This is the mode entered when DIARY is selected from the main Menu.

On entering the mode from the main Menu, the current date and time slot is displayed, together with any entries that may have been made within that time slot.

On entering the mode from the DIARY Menu, the last date and time slot worked on will be displayed, together with any entries within that time slot. A typical display could be:

```
JUL:08:TUE:13.00
(A)LUNCH J.Brown
```

This indicates that a lunch appointment has been made with J. Brown on July the 8th at 1.00 p.m., and that an Alarm call for the appointment has been requested (as indicated by the '(A)').

If the Diary entry extends beyond the screen width, the entry will be scrolled.

It is not possible, in this mode, to stop the scrolling action.

Key operation in the PAGE mode

CLEAR/ON Returns to the main Menu

MODE Selects the DIARY Menu

DEL Deletes the displayed entry.
This is the only mode that allows you to delete Diary entries.

UP Each press reduces the time slot by half an hour. The date remains unchanged.

DOWN Each press increases the time slot by half an hour. The date remains unchanged.

LEFT Each press reduces the date by one day.

RIGHT Each press increases the date by one day.

EXE – or any letter key – selects the EDIT mode to enable a Diary entry to be made. (This mode is indicated by the message 'EDIT' at the start of the second line).
In this mode, the keys function as follows:
LEFT and **RIGHT** cursor keys allow you to move backwards and forwards through your entry.
DEL enables you to delete characters to correct an entry.
CLEAR/ON clears your entry. To abort making an entry, press **CLEAR/ON** again.
EXE or **MODE** saves your entry, and initiates the Alarm Call option.

The LIST option

LIST allows you to browse through your Diary, and is selected from the DIARY Menu. The first entry to be displayed will be the one on or after the latest set date and time slot: if you have not been working on your Diary, this will be the current time and date, as determined by the built in clock and calendar.

Key operation in the LIST mode

EXE Selects the next entry for display, until the END OF DIARY message is displayed. Pressing **EXE** at this point restarts the browsing from the *earliest* entry that has been made.

DOWN Operates exactly the same as the **EXE** key.

UP Similar to **EXE**, but moves *back* through the Diary entries until the first entry is reached.

LEFT/RIGHT Enable any scrolling of an entry to be stopped and started.

MODE Returns to the DIARY Menu.

CLEAR/ON Returns to the main Menu.

The FIND option

This option when selected from the DIARY Menu allows you to locate a specific entry in your Diary, by entering a short 'search clue'. For example, to find when you have an appointment with J. Brown, you could enter 'Bro' or 'rown'.

If no 'search clue' is given, all entries from the last set time and date – or the current time and date – can be displayed.

This mode differs from the LIST option, in that it allows you to transfer to the PAGE mode in order to change or delete an entry.

Key operation in the FIND mode.

Once a 'search clue' has been entered to the FIND prompt, the keys function as follows.

EXE Locates the next entry matching the search clue, until the END OF DIARY message is reached. The next press of **EXE** starts the search again from the earliest entry in the Diary.
Note that any of the alphabetic keys has the same effect as **EXE**.

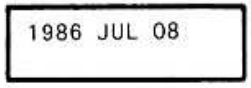
LEFT/RIGHT Enable any scrolling of an entry to be stopped or started.

MODE Switches Organiser from the FIND mode to the PAGE mode, enabling you to change or delete an entry.

CLEAR/ON Returns you to the DIARY Menu.

The GOTO option

GOTO enables you to select a specific *date* in your Diary, in order to make a new entry. When GOTO is selected from the DIARY Menu, the screen displays the current date, in the following form:



The cursor will be flashing over the first figure of the year.

Key operation in the *GOTO mode*.

LEFT/RIGHT Move the cursor over the year, month and date, to

enable you to effect a change.

UP	Each press adds one to the year, month or date, according to the position of the flashing cursor.
DOWN	Each press subtracts one from the year, month or date, according to the position of the flashing cursor.
MODE/EXE	Puts you into the PAGE mode for the selected date. You can now set the time slot for your entry, using the **UP** and **DOWN** cursor keys.
CLEAR/ON	Returns you to the DIARY Menu.

The TIDY option

TIDY enables you to delete all entries in your Diary up to the *latest* PAGE date and time setting. If you have just been using your Diary, this will be the last date and time slot worked on. Otherwise it will be the current date and time, as determined by the Organiser's built in clock and calendar.

The display will be in the form:

```
JUL:07:MON:12.30
DELETE UPTO Y/N
```

When this option is selected, be sure to check the date and time carefully: there is no way to recover Diary entries once they have been deleted.

Key operation in the *TIDY mode*.

Y	Clears *all* Diary entries up to *but not including* the specified date and time.
N	Returns to the DIARY Menu.

Using the Diary

This section summarises the various procedures for using the Diary. Refer back to the appropriate Diary option paragraphs for further details and information regarding the action of the keys within each mode.

The procedures that follow assume that the Organiser has just been switched on and is displaying the main Menu: if you have been using the Diary, return to the DIARY Menu, and pick up the procedure from that point.

Locating a specific date and time

Two methods are available. The first is more suited to dates and times fairly close to the current date and time setting of the Diary.

Method 1:

a) Select DIARY on the main Menu.
You will be in the PAGE mode.
b) Use the **LEFT** and/or **RIGHT** cursor keys to select the required date.
c) Use the **UP** and **DOWN** cursor keys to select the required time slot.
Holding any of the cursor keys down for more than a second will cause that key's operation to repeat automatically, so that you can move quickly through the settings.

Method 2:

a) Select DIARY on the main Menu.
b) Press **MODE** to select the DIARY Menu.
c) Press **G**, to select the GOTO option.
d) Use the cursor keys to select the required date.
e) Press **MODE** or **EXE** to enter the PAGE mode.
f) Use the **UP** and **DOWN** cursor keys to select the required time slot.

Making a Diary entry

a) Select the required date and time slot using Method 1 or 2 in the previous paragraphs.
b) Press **EXE** or the first letter of your required entry.
The word EDIT will appear at the start of the second line to indicate that you are in the EDIT mode.
c) Enter your information.
Your entry can be a maximum of 64 characters, including spaces, and can appear on one line only. (Don't worry if you can't see all of your entry: it will scroll when subsequently displayed). Remember that the **LEFT** and **RIGHT** cursor keys and the **DEL** key enable you to locate and correct errors in your entry.
d) Press **EXE** or **MODE** when you are satisfied your entry is correct.
e) The lower line of the screen will display the message ALARM Y/N (provided that your entry is not in the *past*!).
Press **N** if you do not want an Alarm call for your entry.
Press **Y** if you want an Alarm call. The lower line of the screen will display MINUTES: 15. This tells you that the Alarm will buzz 15 minutes before the event you have entered. To increase the number of minutes warning you will be given (to a maximum of 59 minutes), use the **UP** or **RIGHT** cursor keys. To decrease the amount of warning time you will be given, use the **DOWN** or **LEFT** cursor keys.
f) Press **EXE**.
Your entry will now be displayed, scrolling if it is more than

16 characters long, and preceded by '(A)' if an Alarm call has been requested.

g) You are still in the PAGE mode of the Diary, and can therefore select another date and/or time slot to make another entry.

h) Press **MODE** for the DIARY menu, or press **CLEAR/ON** to return to the main Menu.

Browsing through your Diary.

a) Select DIARY on the main Menu.

b) Press **MODE** to select the DIARY Menu.

c) Press **L**, to select the LIST option.
The first date and time slot containing an entry, starting from the current date and time, will be displayed.

d) Press **EXE** (see *The LIST option* for other keys to use) to display each successive entry in turn, until the END OF DIARY message is displayed. Pressing **EXE** again will cause the earliest entry to be displayed.
Entries longer than 16 characters will scroll: use the **LEFT** and **RIGHT** cursor keys to stop and start the scrolling action.

e) Press **MODE** to return to the DIARY Menu, or **CLEAR/ON** to return to the main Menu.

Locating a specific entry.

a) Select DIARY on the main Menu.

b) Press **MODE** to select the DIARY Menu.

c) Press **F** to select the FIND option.
The screen will display FIND on the top line.

d) Enter three or four letters as a search clue for the entry you wish to find. For example, to find an entry containing the name 'Brown', enter 'Bro'.

e) Press **EXE**.
The first record (after the current date and time setting) that has a match for your search clue will be found: if this is not the one you require, press **EXE** until the required entry is displayed. (If you reach the END OF DIARY message, pressing **EXE** again will take you to the earliest entry with a match for your search clue).

f) Press **MODE** (to enter the PAGE mode) if you wish to change or delete your entry, or press **CLEAR/ON** to return to the DIARY Menu.

Changing or deleting a Diary entry.

a) Locate the entry to be changed or deleted, using the previous procedure.

b) Press **MODE** to enter the PAGE mode.

c) Press **EXE**.
EDIT will be displayed at the beginning of your entry, to indicate you are now able to make corrections.
You can now change or delete your entry, and/or remove or add the Alarm call, by following the appropriate procedure:

Changing the entry:
Use the cursor and **DEL** keys to locate and delete the unwanted information, and insert the new details. Then procede from d) in the section *Making a Diary entry.*

Deleting the entry:

a) Press **CLEAR/ON** to remove the entry.

b) Press **EXE** to complete the deletion.

Deleting/adding the Alarm call:
Press **EXE** to resave the entry. You will now be asked if you wish to set the Alarm: procede from e) in the section *Making a Diary entry.*

Returning to a Menu:
Press **MODE** to return to the DIARY Menu, or **CLEAR/ON** to return to the main Menu.

Tidying up your Diary

a) If you have been working on your Diary, or if you wish to delete entries up to a different date and time slot to that which is current, first locate the entry you wish to delete up to, using either the *Browsing through your Diary* or the *Locating a specific entry* procedure given earlier.

b) Press **MODE** once or twice, to return to the DIARY Menu.

c) Press **T**, to select the TIDY option.
The display will show a date and time slot, and ask whether you wish to 'DELETE UPTO Y/N'.

d) To delete up to (but not including) the displayed date and time slot, press **Y**.
To cancel the deletion, press **N**.
Either way, you will be returned to the DIARY Menu.

Keeping more than one Diary

For most purposes, one Diary will prove to be adequate. However, you may wish to keep separate Diaries, or you may wish to store away one Diary for reference at a future date.

Organiser II provides facilities to enable you to save a complete Diary, either to the RAM area or to a Datapak. A Diary saved in this way cannot be accessed or used by the PAGE, FIND, LIST, GOTO or TIDY options of the DIARY Menu: these operate only on the *current* Diary.

The saved Diary can be restored as the current Diary, but this action completely overwrites the existing current Diary. Consequently, if you wish to continue using the existing current Diary, that would have to be saved for restoring at a later time.

When you save a Diary, you must give it a *name*, to enable you to identify it when you want it restored. Each time a Diary is saved to a Datapak, it occupies 'new' space in the Datapak: if it is saved frequently with the same name, earlier versions will be lost completely and the Datapak will gradually be 'used up'. The Datapak will still be used up if a different name is given to a Diary each time it is saved – but all earlier versions will still be accessible for restoring.

Saving a Diary in Organiser's RAM will *overwrite* any previously saved Diary that has been given the same name. You can erase a Diary saved in RAM, to regain the space it occupied.

Saving a Diary

a) Select the DIARY option from the main Menu.
b) Press **MODE** to select the DIARY Menu.
c) Press **S** to select the SAVE option.
The screen will display the message SAVE A:. This indicates your Diary will be saved to RAM: press **MODE** to display the Datapak you wish to save your Diary to. (If Datapaks are not connected, the **MODE** key will have no effect).
d) You must now *name* your Diary.
The name must be no more than eight characters long, it must start with an alphabetic character, and can contain only alphabetic or number characters (no spaces). It doesn't matter whether you use capital or lower case letters.
Type in the name immediately after the SAVE message. Thus, if you chose the name 'HOME1' and you were saving to the upper Datapak ('B'), the screen display would look like this:

```
SAVE B:HOME1
```

To cancel the SAVE operation at this point, press **CLEAR/ON** twice. You will be returned to the DIARY Menu.
e) Press **EXE**.
Your Diary will be saved, and you will be returned to the DIARY Menu.

To restore a saved Diary

If you intend to carry on using the *current* Diary at a later time, be sure to SAVE it first, using the procedure just given.

a) Select DIARY from the main Menu.
b) Press **MODE** to select the DIARY Menu.
c) Press **R** to select the RESTORE option.
The screen will display the message RESTORE A:. If the Diary you wish to restore is on a Datapak, press **MODE** to select that Datapak ('B' for a Datapak in the upper slot, 'C' for a Datapak in the lower slot).
d) Enter the name of the Diary you wish to restore – the same name that you gave it when it was saved. (It doesn't matter whether you use capital or lower case letters).
e) Press **EXE**.
The named Diary will now be the current Diary, ready for your perusal or work. You will be returned to the DIARY Menu.

To erase a saved Diary

a) Select DIARY from the main Menu.
b) Press **MODE** to select the DIARY Menu.
c) Press **E** to select the ERASE option.
The screen will display the message ERASE A:. If the Diary you wish to erase is on a Datapak, use the **MODE** key to select that Datapak.
d) Type in the name of the Diary you wish to erase.
e) Press **EXE**.
The saved Diary will be erased, and you will be returned to the DIARY Menu display.

To list saved Diary names

a) Select DIARY from the main Menu.
b) Press **MODE** to select the DIARY Menu.
c) Press **D** to select the DIRectory option.
The screen will display DIR A:. Use the **MODE** key, if necessary, to select the required Datapak.
d) Press **EXE**. The name of the first Diary saved to the selected Datapak or RAM will be displayed. Press **EXE** repeatedly to see other Diary names.
e) Press **CLEAR/ON** to return to the DIARY Menu.

2.6

USING THE CALCULATOR

Before you start

You will find the calculator function of Organiser II differs from ordinary calculators. For example, your calculation entry is displayed on the screen, enabling you to go back and correct errors or change the figures for repeated calculations. Also, whereas some calculators have keys 'dedicated' to producing results like square roots, Organiser doesn't – instead, these functions are entered as *words* in an abbreviated form.

Furthermore, you can add to the many functions that are already built in, by writing your own as *programs*.

The ten calculator memories are different, too. They can be used just like any other calculator's memory. But when anything is saved in them, it stays until it is changed, even though the Organiser may be switched off in between. So you can keep the results of calculations for further work at a later date.

When CALC is selected from the main Menu, the keyboard is set automatically for *numeric* inputs: that is, all the characters printed above rather than on the alphabetic keys. The actual numbers are easily identified: they are on an area of different background colour to the rest of the keyboard.

The basic mathematical operator keys are as follows:

+	Add
–	Subtract
*	Multiply
/	Divide

If the multiplying symbol, *, is entered twice, it means 'raise to the power'. Thus, **3**2** means 'three raised to the power two' – or 3^2.

The brackets (and) can be used as in ordinary calculators, to group together parts of a calculation which need to be worked out first. Organiser II lets you have brackets within brackets – as many times as you wish.

If a number is prefixed by the $ symbol, that number is considered by the Organiser to be *hexadecimal*. Thus **$20** is the hexadecimal number 20 – which is 32 in decimal. Before you panic, hexadecimal is just a numbering system used by experienced computer programmers, and it is a very useful feature for them to have on a calculator. If you don't understand it, don't worry.

The calculator will also recognise the symbols < , > and = as 'less than', 'greater than' and 'equals' *tests* respectively. These *tests* produce a result of **–1** if true, and **0** if not true. Thus a 'calculation' such as **4=4** would produce the seemingly crazy result of **–1**, signifying that it is indeed true that four equals four.

Again, if all this is new to you, don't worry about it: the use of these symbols is important when programming to test the results of a calculation, and will be explained fully in the programming section of this book.

Finally, the **%** symbol: this does not produce 'percentages'. Like the **$** symbol, it has a special significance in computers and is mainly used in programming. However, the calculator will recognise a **%** symbol *in front of* a character as meaning *the pattern number for that character*. These pattern numbers are explained in Chapter 1.2, under the heading *Storing characters*. The use of **%** as a prefix is consequently limited to ascertaining the pattern number for a character in Organiser II.

Fixing the decimal point

Organiser performs its calculations to 12 significant figures, rounding up the last figure if necessary. Thus, a calculation such as '7/9' will produce the result '0.777777777778'.

You may not always want so many figures after the decimal point. If you are working with money calculations, for example, you will want only two figures – representing pence.

Organiser II lets you determine how many figures you have after the decimal point. With Organiser in its Calculator mode, so that the screen looks like this



type in FIX= followed by the number of decimal places you want. Remember to hold down the **SHIFT** key while you type in the letters 'FIX'.

Now, any calculations you make will have only the number of decimal places in the answer that you entered after 'FIX=', to a maximum of 12.

The number of decimal places you set will be remembered by Organiser even when you switch off. So the next time you come to make calculations, they will be to the same number of decimal places that you set previously. You can of course reset the number at any time. To revert back to 12, you need only type in FIX=. If Organiser can't find a number after the = sign, it uses its 'default' value of 12.

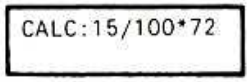
Entering a calculation

You enter a calculation into the Organiser just as you would an ordinary calculator. The difference is, you can see what you are doing, because your calculation is displayed as you enter it, on the top line. If your calculation takes up more room than the top line permits, its scrolls to the left.

When you have entered your calculation, you execute it by pressing **EXE** (*not* the = key).

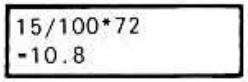
The answer will appear on the second line, following an = sign, and your calculation will remain on the top line – scrolling if necessary. The CALC message will no longer be displayed, indicating that Organiser is displaying a result, and is not ready to receive another calculation at the moment.

For example, suppose you wished to find 15% of 72. Your calculation would be entered as



Note that Organiser puts 'CALC' on the screen – you don't have to enter it. If CALC *isn't* displayed, you cannot enter a calculation.

When **EXE** is pressed, the display becomes



If you had FIXed the number of decimal places to two, the answer would read 10.80.

Now, supposing you wished to find 15% of some more figures. No problem: press **EXE** again, and the display will revert back to the way it was when you finished entering your calculation, with the cursor at the end of the line. You can now use the cursor keys to go back over your calculation, and the **DEL** key to make changes. In this instance, you would remove the '72' and replace it by the next figure, then press **EXE** for the new answer.

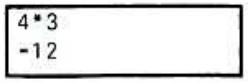
EXE is not the only key that will return you to your calculation: other keys can be used, too, with different results. Here is a summary:

EXE	Returns to calculation, cursor at end.
DOWN	As above.
RIGHT	As above.
UP	Returns to calculation, cursor at beginning.
LEFT	Returns to calculation, cursor under last figure (or symbol).

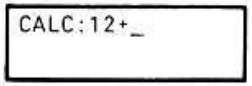
DEL	Returns to calculation, the last digit or symbol is deleted, and the cursor is at the end.

You can also continue with a calculation, using the answer obtained as the first value, by pressing any of the mathematical operators – +, –, * or /. When any of these is pressed, the answer replaces your original calculation, and it is followed by the symbol for the pressed key.

For example, if this is the display after a calculation



pressing the + key will result in:



The underline indicates the cursor is positioned for your next input.

To clear an answer *and* the calculation, press **CLEAR/ON**.

To return to the main Menu, press **CLEAR/ON** while the screen is displaying just CALC on the top line.

Very large numbers

The Calculator handles very large numbers by using a system called *scientific notation*. This always looks more frightening to the uninitiated than it really is.

If the number to be represented on the display has more than 15 digits before the decimal point (of which only the first *12* are significant – the rest must be zeroes), Organiser displays the number *as a decimal number* – with just one digit before the decimal point. This is followed by an 'E+' and the number of *places* the decimal point must be shifted.

For example, if the answer to a calculation happened to be (or if you simply enter)

1234567890120000

it would be displayed as

1.234567890E+15

when **EXE** is pressed. The 'E' indicates that scientific notation is being used. The +15 indicates that the decimal point must be shifted *15 places* to the right of its *current* position. Note that, in the example just given, the '12' before last few zeros is lost: in scientific mode, Organiser works to ten figures, two figures being needed to determine the decimal point shift.

If you're working with astronomical figures, you can enter calculations in scientific notation.

For example, 123456000 in scientific notation is written as 1.23456E+8. You could multiply this by 2 by entering

1.23456E+8*2

In this instance Organiser would display the answer as

246912000

because it goes into scientific notation only when it has to.

The order in which calculations are made

Just like you, the Organiser has to work out a calculation a bit at a time. It always looks to see if there are any brackets, and works out the bits in brackets first. If there are brackets within brackets, the calculation within the 'innermost' brackets gets tackled first.

Thus, if the calculation is

4+3–(3/(5+6))

the '**5+6**' part of the calculation will be evaluated first, then the result is divided into 3, and finally the rest of the calculation is made.

Once Organiser has decided which part of a calculation should be done first, it then has an order of tackling the mathematical operators. It first looks to see if any of the values are *negative* (rather than being *subtracted*), and sorts that out. For example, you could enter

4*–3

Here, the calculation is to multiply 4 by –3 (=–12): it is not going to *subtract* 3. This kind of 'minus' is called a *unary minus* and it is indicating that the 3 is negative. It gets top priority in the Organiser's sequence of working out a calculation.

The next priority is given to 'powers' – entered in the Organiser as '**'. These are evaluated from right to left. Thus for a calculation such as

32**4**

first the 2 would be raised to the power of 4 (=16), then 3 would be raised to the power of the answer, **16**.

Note that Organiser has a special routine for working out 'powers' – which provides an extremely high degree of accuracy. However, for a simple power calculation such as 2^3, (entered as **2**3**, and meaning '2x2x2'), the answer is not *precise*. For the example just quoted, Organiser would give the result as 8.00000000005 – instead of just 8. This special routine is necessary for Organiser to be able to handle really tricky powers.

Next in order of calculation precedence comes any multiplication and division. Then comes addition and subtraction and finally come the 'comparison' operators – <, > and =.

Thus, in a calculation such as

4+3*10/2–6*5

the sequence of calculation is first the multiplications and divisions – **3*10/2** (=15) and **6*5** (=30). Then these results are used in the additions and subtractions – **4+(15)–(30)**, to give the answer **–11**.

If, in this example, you wanted the **10** to be divided by the result of **2–6*5**, and worse, you wanted the **6** to be subtracted from the **2** *before* it is multiplied by the **5**, then you would have to tell Organiser the order of events by enclosing within brackets the items you want handled first. Thus, to achieve the required result under these conditions, the calculation would have to be entered as

4+3*10/((2–6)*5)

The calculating sequence this time is:

1) The innermost bracket: **2–6** = –4
2) The next bracket: **(–4)*5** = – 20
3) The multiplication and division: **3*10/–20** = –1.5
4) The addition: **4+(–1.5)** = 2.5

As you can see, working the calculation out his way produces a completely different result: **2.5**.

If you are ever in doubt regarding the order in which your calculation will be made, enclose the parts *you* want calculated first or together within brackets. It doesn't matter how many sets of brackets you have – as long as there is always a closing bracket for an opening bracket. If you do happen to miss out a bracket (or make some other mistake), Organiser will tell you the type of mistake you have made and will ask you to press the **SPACE** key. Your calculation will then be displayed again, with the cursor marking the point where Organiser failed to understand what it should do.

Very helpful.

Using the Calculator memories

It has already been mentioned that the calculator has ten memories, and that their contents are kept even when Organiser II is switched off. These memories have another useful feature – they can be accessed from your programs. Thus you could write a bank-balance/cheque program that uses one of the calculator memories to store what you think the bank should have on your behalf. You could then look into that memory at any time to check the balance, without having to use the program.

The ten calculator memories are *always* identified by the same 'names' – M0 to M9, inclusive – whether you're accessing them from the calculator or from a program. Here's how to use them.

To store in a calculator memory

a) First, there must be an *answer* on display – that is, the second line must have the = sign followed by a number, usually as the result of a calculation. If you wish to store a number without making a calculation, enter that number after the word CALC, then press **EXE**: the number will then be reproduced on the second line.

b) Press **MODE**.
The top line will display M: PRESS 0–9

c) Select which of the calculator memories you wish to use, by pressing the appropriate number (0 to 9). The top line of the display will change to show the memory you selected, followed by +,–,**EXE**,**DEL**.

d) You now have five options:
   1) To add the number on the second line to what is already in the selected memory, press +.
   2) To subtract the number on the second line from what is already in the selected memory, press –
   3) To just save the number on the second line in the selected memory, overwriting anything that is already there, press **EXE**.
   4) To clear the contents of the selected memory to zero, press **DEL**.
   5) To call the whole thing off, and go back to the calculation and its result, press **CLEAR/ON**.

To recall a number from memory

Simply enter its name.
This can be done within a calculation: **4.2*M5**
Or, if you just want to inspect its contents, enter the name alone and press **EXE**.

The built-in functions

Organiser's programming language has a number of *numeric* functions, any of which can be used in calculations. Some, however, are more suited for use in programs – 'HOUR', for example, is one such function. This gives the current hour from Organiser's built in clock: useful for certain kinds of program, but not of great value when making calculations.

Most of the functions suitable for calculations are intended for scientific and engineering type of work. A list of these follows, together with a brief description. You will find a complete list of the other numeric functions within the programming Chapters of this book.

To use a function, you simply 'name' it in your calculation, followed, in most instances, by the value to be 'worked on' enclosed in brackets. This value could be another calculation. For example, to multiply the logarithm of 5/9 by 4, you would enter:

4*LOG(5/9)

To perform this calculation, Organiser would work out **5./9**, find the LOGarithm of the answer, and multiply *that* result by **4**.

The main functions are as follows:

ABS(no.)	Gives the number as a positive value, even if it is negative. Thus 'ABS(4–6)' gives **2**, not **–2**.
ATAN(no.)	Gives the Arctangent of the number, in radians.
COS(no.)	Gives the COSine of the number, which represents an angle in radians.
DEG(no.)	Converts the number, which represents an angle in radians, to degrees.
EXP(no.)	Raises the arithmetic constant 'e' (2.178) to the power of the number in brackets.
INT(no.)	Gives the part of the number *before* the decimal point – the *integer* of the number. Thus 'INT(10.3)' would give **10**. Negative numbers are rounded down, so 'INT(–7.6)' would give **–8**.
LN(no.)	Gives the logarithm of the number to the base 'e'
LOG(no.)	Gives the logarithm of the number to the base 10.
PI	Gives the value of pi (3.14159265359...)
RAD(no.)	Converts the number which represents an angle in degrees, to radians.
RND	Gives a random value between 0 and 1.
SIN(no.)	Gives the SINe of the number, which represents an angle in radians.
SQR(no.)	Gives the square root of the number.
TAN(no.)	Gives the TANgent of the number, which represents an angle in radians.

Remember that in all instances where a *number* is needed, it can be either a specific value, or a calculation that results in a value. Also,

the value or calculation must be enclosed within brackets.

Example: To find the result of four multiplied by the square root of 256, the display would look like this:

```
CALC:4*SQR(256)
```

Don't forget to press the **SHIFT** key when entering *letters*. This calculation will be made as soon as you press **EXE**.

Using your own functions

There may be some calculations that you need to make quite frequently – adding or deducting VAT, miles per gallon, loan repayments, and so on. One of the purposes of being able to program Organiser II is so that you can undertake such calculations with the minimum of effort – that is, by simply entering the necessary figures, without all the *calculation* part.

Furthermore, you can arrange your program so that it prompts you for the figures it needs to make the calculation. All of this is discussed in Part 3.

In addition to *complete* programs, there may be *functions* – like the scientific ones given in the previous section – that you wish to use when making a calculation. A very simple one, for example, could convert litres to gallons.

Whether it's a complete program or a function, you would 'call' it for use in the Calculator mode the same way – and this is almost the same as the way you 'call' up the built-in functions.

There is just one difference. For your own functions or programs, you must add a colon (:) after the *name* part.

A couple of examples will help to clarify the procedure. Let us say, first, that you have written a *function* to convert litres to gallons – so that you can see how many miles to the gallon you are doing even though you pumped in *litres*. And let us say that you called this function 'GAL'. When such a function is used it needs a value to work on – in this case, the number of litres. This value must be enclosed within brackets (just as a value must be enclosed in brackets for most of the built in functions).

The way to work out miles per gallon, as you know, is to divide the number of miles by the number of gallons. But you pumped in litres, so when making your calculation, you would enter

miles/GAL:(*litres*)

You would use actual numbers in place of *miles* and *litres* of course. Note the colon after the *name* of your function. That tells Organiser

II not to waste its time looking through its own functions – it must look through RAM and the Datapaks for the one you will have written, called 'GAL'. Note too that the brackets come *after* the colon.

Organiser will work out the part in brackets – if it needs working out, then use your program to convert the answer into gallons. It then divides *that* answer into the miles figure to give you the number of miles-per-gallon.

Let us now take another example, this time for percentages. It could be that you frequently need to know what percentage one number is of another number. You would therefore write a very simple little program, which needs *two* inputs. Your program would take the first number, divide it by the second and then multiply the answer by 100 to give the required percentage. This is, of course, a very simple calculation, and wouldn't take very long to enter as a straight calculation. Nevertheless, it will serve to demonstrate the principle. Having written your program, which we'll say is called PC, you could use it to find, say, what percentage 37 is of 96 by simply entering

PC:(37,96)

and this could be just a part of a larger calculation.

Notice that the two numbers are separated by a *comma*. That is important. Equally, it is important that you enter your numbers in the right order – to match the way the program is written: the first number divided by the second.

You can write programs to accept almost as many inputs as you want – certainly as many as you're ever likely to need.

Within the brackets part of one of your functions, you could include *another* function. Taking the two examples given here, you could work out what percentage a number of gallons is against a number of litres. Let us say you want to know what percentage 34 gallons is of 200 litres (pretend! pretend!). This would be entered as:

PC:(34,GAL:(200))

Organiser would first use your 'GAL' program to convert the litres to gallons. Then it would use your 'PC' program to evaluate the percentage. It works, always, from the innermost brackets outwards remember.

One final point. The calculator works in *floating point* arithmetic. That means it expects numbers to have a decimal point in them somewhere, even if it isn't shown (e.g. '10'). When programming, you can identify numbers that will never have a decimal point – they're called *integers*. If you write a program that expects an *integer* to work on, you will get an error if you try to use that

program from the calculator. The message will come up **TYPE MISMATCH**.

Organiser is telling you that you are trying to mix two types of number – *floating point* and *integer*. As the two types of number are stored differently inside the Organiser, it's a bit puzzled as to what it should do. (Why have two types of number? Because *integer* numbers take up much less memory space, and calculations with them are many times faster).

As always, there is a solution to the problem. Two, in fact. You can either make sure that any program you are going to run from the calculator requires only *floating point* numbers to be passed to it. Or you can convert the number passed to your program to an integer type, using the INT(number) function. With the second method, you would of course lose any fractional part of a number passed, and this could cause calculation errors. 'INT(5/2)', for example, results in the value of **2**, not **2.5**.

If the 'GAL' program needed an integer input, you would have to write 'GAL:(INT(33))', *even though the number you are passing doesn't have a decimal point in it.* The INT(number) function converts the *way* the number is stored – and hence the way Organiser can use it.

When things go wrong

When Organiser comes up against something it doesn't understand, or if it finds an error in something you've entered, it will tell you – and usually, it will tell you *where* it has found the mistake. For example, if you're making a calculation that uses several programs that you've written, and there's an error in one of them, Organiser will not only announce the type of error it found, it will also tell you which program has the error.

It goes even further. When the **SPACE** key is pressed – to clear the error message from the screen – Organiser will often (but not always) display the place where it found the error, with the cursor marking the point. How close to the location of the error it gets really depends on the type of error that has been made.

When making a calculation that uses a number of brackets, for example, you may get the message **MISMATCHED ()'s** if there are too many 'closing' brackets. On pressing **SPACE**, the calculation will be displayed with the cursor under what Organiser believes to be the offending bracket – the one it couldn't understand. If you have too many 'opening' brackets, the message would be **SYNTAX ERR** and on pressing **SPACE**, the calculation would be displayed with the cursor at the point where Organiser believes there should be a bracket.

The abbreviations used in error messages likely to be met when making calculations are

ARGS	Arguments – the numbers passed to a function
ERR	Error
FN	Function
PROC	Procedure – another name for a program.

Handling *programming* errors is discussed in Chapter 3.17.

PART 3

Programming Organiser II

There are three aspects to programming. First, there is the *principle* involved – how to set about it and plan what you want to do. Second there is the *mechanical* process of actually entering your program, using the facilities provided. Third there is the programming *language* – the special instructions that allow you to achieve the result that you want. The first two are dealt with in the first two Chapters of this part of the book: the remaining Chapters deal with the programming language.

3.1

THE PRINCIPLES OF PROGRAMMING

What is programming?

Quite simply, programming a computer means nothing more than giving it a series of instructions to perform a particular job. The program, once written, tested and *saved*, can be used over and over again without having to re-enter the *instructions* repeatedly. All that has to be entered is the *information* the program needs to work on.

Thus, for regular tasks, a program is a great time and effort saver.

The program is called the *software*, and the computer with all its circuitry is called the *hardware*. Using a cassette recorder as an analogy, the *music* one records is the software, the recorder itself is the hardware. The tape *cassette* is the medium for carrying the software: with Organiser II, this medium is built in (the ROM and RAM memory) – with provision for adding more (the Datapaks).

Programs have already been written and saved into your Organiser II (in ROM), to provide you with the built-in facilities such as the Diary and Alarm Clocks.

The interesting thing about programming is that, if you were to give each of 100 programmers the task of writing a program to handle a particular job on one particular computer, you would be faced with 100 different programs. On the surface, they would all perform the same job. But if you were to examine the actual *instructions*, they'd almost certainly be completely different.

Some will require more memory space than others. Some will operate faster when used. Some will be well planned and easy for other programmers to understand. Some will be a hotch-potch of instructions. Some will be more 'user friendly' – that is, will give lots of prompts and help to guide the program *user*.

The reason why all the programs would be different is everyone has their own ideas on how to tackle the requirement.

The truth of the matter is there is no *right* and *wrong* way to write a program: only good ways and bad ways. If it works, it's 'right'. If it doesn't, it's 'wrong'. Later, when you become more experienced, you will look back over your early efforts and think "I could have written that better – to take up less space, to work faster, to be *neater* and more understandable, to be more user friendly".

When you write programs for yourself, you can make them as 'friendly' as you wish: making them *shorter* and less *space cons-*

uming comes with skill, practice, and a deep understanding of the *language* used.

Don't be put off by that word *language*: when you use a calculator, you are in effect programming it as you go along, using the language of mathematics: '+' to add, '–' to subtract, and so on. The difference between a calculator and a computer is that a computer has many more instructions, and it lets you save your instructions for future use.

The programming language built into Organiser II has about 120 different *words*, each one being a different instruction for Organiser to do something. That may sound a lot compared with the number a calculator has, but when compared with a foreign language, you can see that it shouldn't take too long to learn them.

In any event, *you don't need to know all the words in order to write a program.* Just as with a foreign language, you can get by with only a handful. (Many of the words in Organiser's programming language you may well never use anyway). The more you know, of course, the more 'fluent' you will be as a programmer, and the better your programs will be.

In addition to the program words, you need to know the *grammar* and *punctuation* of the language – better known as the *syntax*. On this score, computers can be unforgiving. They have been 'told' to expect their instructions to be given according to specific rules. If these rules are not obeyed, they don't know what to do, and will either stop to report a programming error, give the wrong results, or 'crash out' – that is, appear to be completely out of control and, in the vernacular, 'doing their own thing'.

There is one thing a program can *never* do – and that's harm or damage the *hardware* of the computer.

Defining the requirement

The first thing to do when writing a program is to define the requirement completely, and as precisely as possible. Not in your head – on paper. Make sure you have covered all aspects of the requirement, including how you want to use the program.

If you want to use the program as a *function* when making calculations, your requirement may well be no more than a formula. Suppose, for example, you wished to write a *function* for converting litres to gallons, or for calculating the percentage one number is of another.

In both cases, a simple formula is used. Write the formula down, so that you can see what the program has to do. Thus, for *Litres to Gallons* you would put 'Number of Litres x 0.22'. For the *Percentage a first number is of a second number* function you would put '100 x First number/Second number'. Notice that the 'unknown quantities' are defined as words – to be replaced by figures when the program is actually used.

If you want the program to perform a more complex task – to look after club membership details, perhaps, or to help budget your domestic accounts – the requirement will need to be more explicit. You will need to set down quite clearly what the program has to do, what you expect from it, and what information is needed to provide the required results.

Let us take an example for demonstration purposes. We will plan a program to calculate the materials required to decorate a room, whether we choose to paper the room, or emulsion it.

First, let us set down the program requirements – the *brief*. We want the program to tell us:

a) How many rolls of wallpaper we'll need.
b) How much emulsion we'll need for the walls.
c) How many rolls of ceiling paper we'll need.
d) How much emulsion we'll need for the ceiling.

Now we must set down the information that we're going to need in order to make the necessary calculations.

a) The length of the room.
b) The width of the room.
c) The height of the room.
d) The length and width of the wallpaper.
e) The coverage of the emulsion.

Question: Do we want to take into account windows, alcoves and doors? At the end of the day, window and alcove areas tend to cancel each other out – with the error falling on the side of too much of the material, rather than too little. So we will ignore them. (Allowances can always be made when using the program).

Question: What units of measurement are we going to use – feet and gallons, or metres and litres. Or a mixture? If we want both, then we'll need to add to our program specification a way of knowing which type of unit is being entered, and checks to ensure that all the units used in any calulation are of the same *type*. For demonstration purposes, we shall restrict our program to feet and gallons: to use metres and litres, the figures can be converted *before* they are entered into the program.

Now we need to put down the calculations that will be necessary. First, to evaluate the amount of emulsion needed, we simply divide the *area* by the *coverage* per gallon – both being measured in square feet. Thus:

Gallons required = area/coverage

We can go further: the area of the ceiling is simply the length of the

room multiplied by its width. The area of the walls is the area of the long wall *twice*, plus the area of the short wall *twice* (ignoring windows), or

2*length*height + 2*width*height

This can be 'simplified' if we consider the area of one long wall and one short wall together – height*(length + width). Our room has 'two' of these, so we can write

2*height*(length + width)

As you can see, this is a shorter way to write down the required calculation. Now, for the wallpaper calculation. Wallpaper rolls are about 33 feet long and between 1ft 8 inches and 1ft 9 inches wide. Assuming we want only whole lengths on the wall, we can get the number of pieces that can be cut from a roll by dividing the length of the roll by the room height, and ignoring any 'remainder'. We can get the number of pieces required to go round the room, by dividing the distance round the room by the roll width. Thus:

Pieces per roll = integer (roll length/room height)
Pieces required = room perimeter/roll width

'Integer' in the first calculation simply means ignore any fractional part the answer may have.

The number of rolls needed will be the total number of pieces required, divided by the number of pieces that can be cut from a roll. (These calculations ignore any pattern drop there may be on the paper – this will tend to be allowed for by the fact that we've included the windows in the measurements). If we take the roll length to be 33 feet and the width to be 1ft 9 inches (1.75 feet), the calculation becomes

Rolls required = (Room perimeter/1.75) / integer(33/room height)

The perimeter of the room is, of course, 2*(length + width).

This will give you some idea of the information that you need to set down at the initial stage of planning your program. Without a full *specification* at the outset, you will find yourself trying to add bits at a later date – not impossible, of course, but the more you try to add, the more difficult it can become, and the program will be more prone to error when it is first used. Also, by setting out any calculations that are needed, you will be able to see *common* elements.

Plan the program flow

This stage – called *flowcharting* – is rather like drawing up the

blueprint for your program. It defines the *way* you are going to provide the program requirement, step by step. It's a stage that many programmers tend to ignore. A small proportion of them can afford to – they can call upon previous experience to know what they're doing. The rest believe they belong to the small proportion, and find their programs don't run the way they should first time round. They then waste time *debugging* their program – finding out why it doesn't work, and correcting it.

(*Debugging* comes from the days when computers used valves and relays: insects were attracted to the light emitted by valves, sizzled themselves on the hot glass, and inconsiderately dropped as corpses into the delicate circuitry, causing all kinds of problem. It is alleged that such an event occured to one of the computer pioneers – who found a moth stuck between contacts of a relay. Thus, if anything ever went wrong, it was blamed on a *bug*).

For you, the beginner, the message is clear: prepare a flowchart or at least a plan of action, however unecessary it may seem to be at first.

A flowchart sets out the *sequence* of instructions that the program must follow. Quite often, a point will come in the sequence when there is a choice of routes to be taken. In our 'Decorarting materials' program, for example, if we decide on wallpaper, the program will take one route, if we decide on emulsion, it will take another route. A *test* or an answer to a question is needed to ascertain which route should be taken.

The sequence of events to calculate what percentage a first number is of a second number is very simple:

Get the first number
Get the second number
Work out the answer
(Put the answer where it is wanted)

This is a straight plan, with no tests. But supposing we said that our program should *always* find what percentage the *smaller* number is of the *larger* number – so that it didn't matter what order they were entered into the program. The flowchart could then be as shown in Fig.3.1.1 (a) or (b).

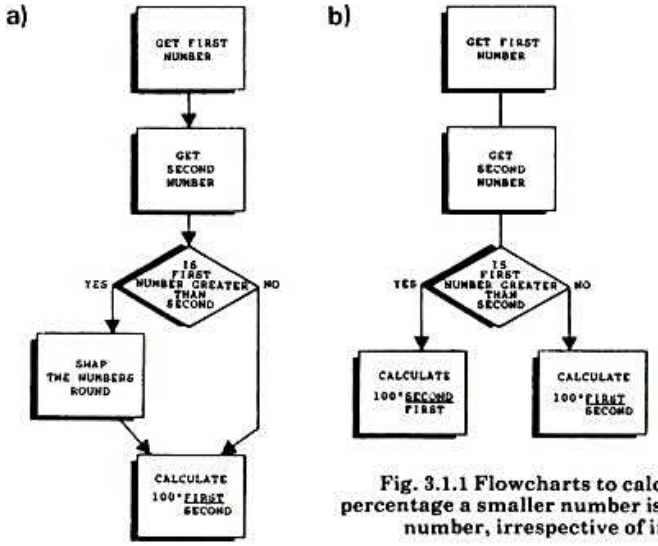


Fig. 3.1.1 Flowcharts to calculate what percentage a smaller number is of a larger number, irrespective of input order.

Both the flowcharts show that having obtained the two numbers needed to perform the calculation, a *test* must be made to see whether the first number entered is larger than the second number entered.

If it isn't larger, we can make the calculation

100 * First number / Second number

If it is larger, then *in order to use the same part of the program for the calculation* (Fig 3.1.1 (a), we must change the two numbers over – make the first number the second, and the second number the first.

The alternative to swapping the numbers round is to perform a different calculation if the first number entered is larger (Fig 3.1.1 (b)):

100 * Second number / First number

Already, we have two ways to solve the requirement! You can understand now, perhaps, how different programmers arrive at different solutions to the same problem. Both methods will work, so neither is 'right' or 'wrong'. Furthermore, when we get to actually writing out the program instructions, there will be even more alternatives – all of them 'right', provided they work.

The important point to remember is that, with a flowchart, the program sequence is made abundantly clear.

With large programs – or even moderately sized programs – the first flowchart may simply outline the broad *structure* of the program: each segment of it can then be broken down into its own flowchart, and the segments of these broken down even further if necessary. For example, a flowchart for an 'arcade' type of game (such as the infamous *Space Invaders*) could be as shown in Fig. 3.1.2.

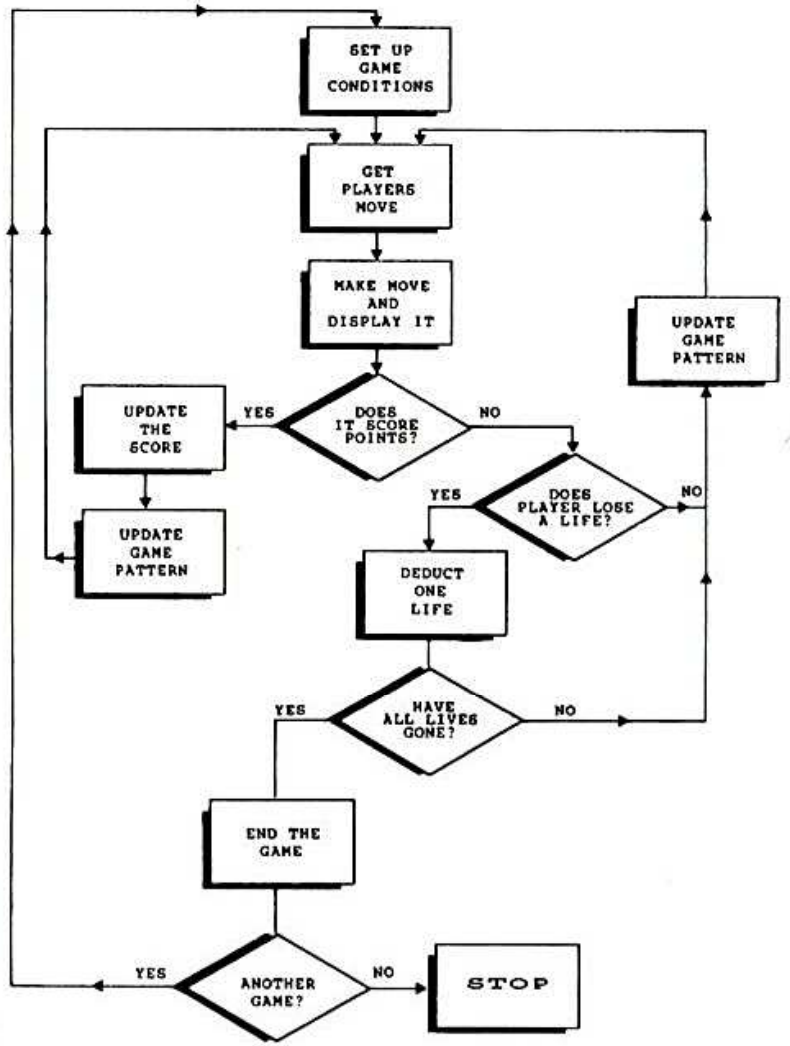


Fig. 3.1.2 Typical Arcade game flowchart

Most of the 'boxes' in this flowchart would need a further flowchart to define more closely the actual programming sequence: for example, the box to 'Update the Score' could well become the flowchart shown in Fig. 3.1.3.

Notice how the game keeps 'looping' back to the 'Get Player's Move' box until all the 'lives' are lost, and then there's the option to go and play the whole thing again. Notice, too, that one box – 'Update Game Pattern' – appears twice: there will be no need to write this part of the program twice. It can be written as a separate program, and *called* when required from the main Game program.

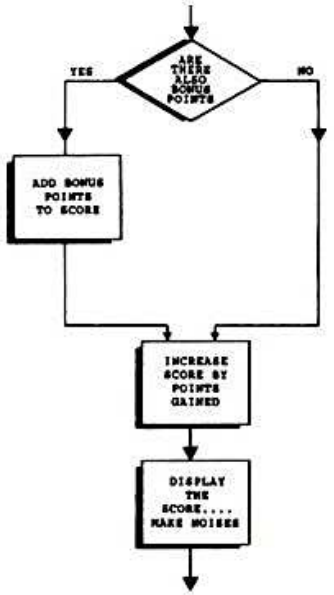


Fig. 3.1.3

'Update the Score' flowchart

In fact, the way the programming language in Organiser II works means that *every* small 'chunk' of the overall program can (and should) be written as a separate program – called a *procedure*. We will come back to this in more detail later. It means that the main program – which follows the framework devised for the master flowchart – virtually just 'calls up' all the sub-programs, and these in turn call up smaller chunks, until you get to the short procedures, which do all the work. (See also Chapter 1.4 – *Obeying Program Instructions*).

Not all programs will need lots of parts, of course – some will be complete in themselves. The evaluation of percentages discussed earlier would need only one *procedure*. Once written, this procedure could be called by any other procedure – and so become part of a larger program.

This is, in fact, one of the most valuable features of the Organiser's programming language: it allows you to build up a library of often used procedures, so that writing larger programs becomes a simpler task. You can, in effect *add* words to the programming language to suit your own needs.

Now, what about a flowchart for our demonstration program – to calculate the quantity of materials needed when decorating. It's worth spending a few minutes to see if you can devise an *outline* flowchart for yourself. One possible solution is shown in Fig. 3.1.4. Notice how, in this solution, the 'questions' that need to be answered by the program user – 'ceiling or walls' and 'emulsion or paper' – are grouped together: when we program this part, we shall cover them all in one procedure, and *store* the answers for subsequent use.

Notice too how, after the required calculation has been completed, provision is made to

a) repeat the program for the same room without having to re-enter the dimensions (you may wish to test out the alternative material, or get the calculation for the other part of the room).
b) repeat the program for a new room.
c) finish using the program.

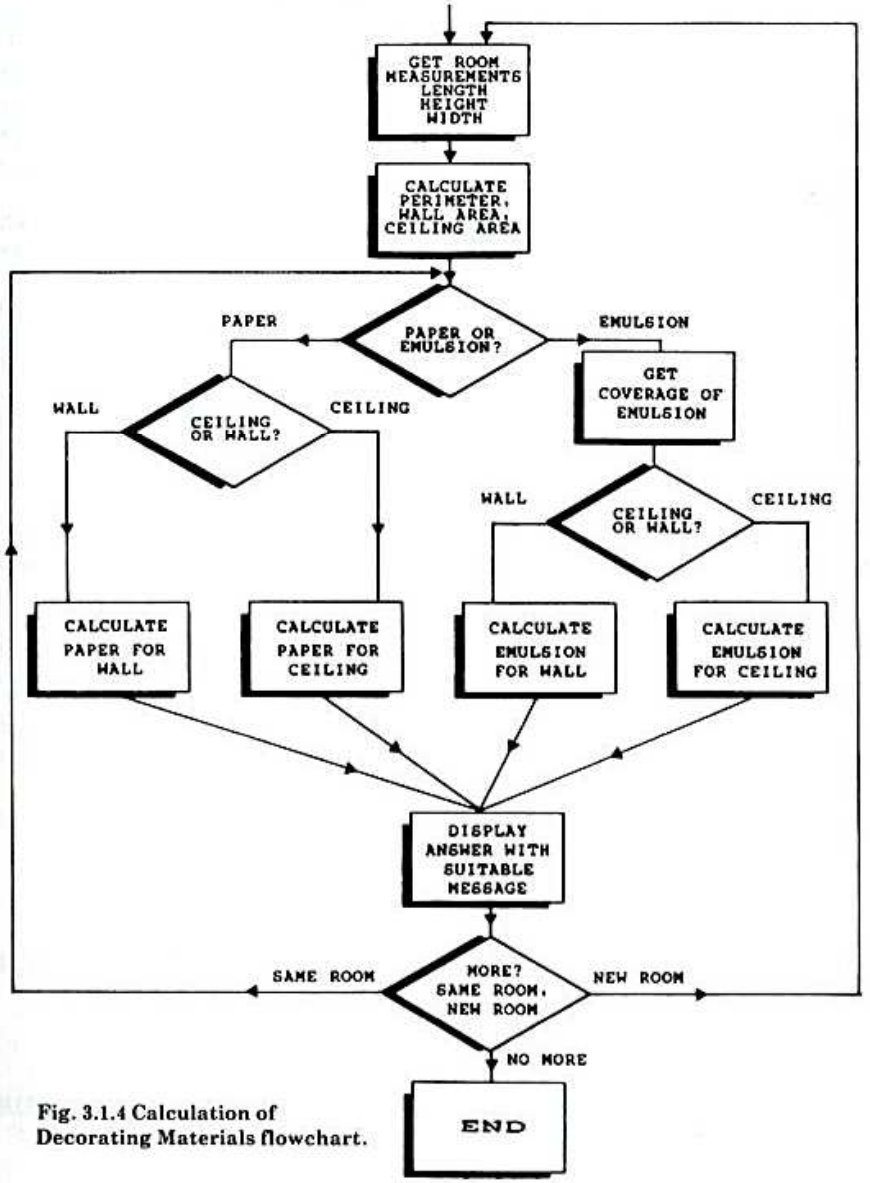


Fig. 3.1.4 Calculation of Decorating Materials flowchart.

Summary

In this Chapter, we have seen that before starting to write out program instructions, it is strongly advised that you take certain steps:

a) Prepare a specification of all the things you want your program to do, with as much detail as possible.

b) Prepare a master flowchart, and if necessary 'sub' flowcharts: when you come to actually writing out the instructions, this will make your life very much easier – and will help to eliminate *bugs*.

Naturally, at this stage of learning to program, it will all seem a little mysterious to you. You will probably find it difficult to define your program requirements, and even more difficult to prepare a flowchart or to 'structure' it to achieve the desired results.

Take heart. When you understand what the actual instructions are and know how to write them into the Organiser, you will understand better what is required when making your initial plans. Come back to this Chapter after you have learned some of the language, and it will undoubtedly make far more sense.

3.2

USING THE PROGRAM MENU

The Programming options

It would make for very slow and cumbersome reading if, throughout the Chapters on how to use the language to write programs, explicit instructions were given *every* time to explain how to enter a program, save it, test it and so on.

It would be rather like giving someone directions to drive to a particular town, and telling them where to change gear *en route*, when to stop for a break, and so on.

Consequently, this Chapter is devoted entirely to how to use the options provided from the Programming Menu. Refer to it as and when necessary: gradually, the *mechanical* process of writing and proving a program will become second nature – like driving a car – and you will be able to concentrate more and more on the actual *programming* process. Instructions will be given where necessary in the language Chapters, particularly the earlier ones, to save you too much turning back and forth.

The Programming Menu is selected by PROG on the main Menu, and offers the following options:

```
EDIT LIST DIR
NEW RUN ERASE
COPY
```

To return to the main Menu from the PROG Menu, press **CLEAR/ON**. Briefly, the options provide facilities as follows:

NEW — Allows you to start work entering instructions on a *new* program procedure, that is, one that you haven't worked on before. When you have finished entering instructions, you can save your program ready for RUNning or EDITing.

EDIT — Allows you to go back and do some more work on a program procedure that you already *saved*, perhaps to make changes or corrections. When you have finished your work, you can save it for RUNning and/or EDITing again.

RUN — Allows you to execute a previously *translated* and saved procedure or program, either to test it or to actually use it.

ERASE Allows you to delete a procedure from wherever it has been saved (RAM or a Datapak).

COPY Allows you to copy a procedure or procedures from one location to another – from RAM to a Datapak, for example.

DIR Enables you to see a list of all the procedures you have written and saved in RAM or on a Datapak.

LIST Allows you to list the instructions that you have entered to a peripheral device such as a printer. (You can, of course, see the instructions when you choose the EDIT option).

The NEW option

You would choose this option whenever you wish to *create* a new program procedure.

a) Select NEW from the PROGramming Menu.
The screen will display NEW followed by the letter indicating the last location worked on – A:, B: or C:. It is strongly recommended that you select RAM (A:) when writing and editing programs (use the **MODE** key to change the location).

b) You must now enter the *name* of your program procedure.
The rules governing this name are:
1) It must be no more than eight characters *in total.*
2) It must start with a *letter.*
3) It can comprise only letters or numbers, except for the last character which must be **$** or **%** if the procedure is to 'return' a *character string* or an *integer.* (See the examples below).
4) It must be different from any other name you have given to a program procedure, (and any of the main Menu option names, if you wish to install your program there).
Choose a name that will help you to identify the purpose of the program procedure at a later date.
You can edit the name by using the **LEFT** and **RIGHT** keys to postion the cursor, and the **DEL** key or **DEL** plus **SHIFT** keys to erase unwanted characters.

c) When you are happy with your entry, press **EXE**.
The screen will display the name you entered, followed by a colon.

d) If a *parameter* is to be passed *to* your procedure from the Calculator or from another procedure, enter the *name* you intend to use for that parameter within *this* procedure, within brackets. If more than one parameter is to be passed to the procedure this way, separate each from the previous one by a comma. The kind of value or information being passed in *must* be followed by its *type identifier.*

Example 1: If you were writing the procedure for percentages discussed in the previous Chapter, which is to 'return' a *floating point* value, and you chose the names for the First and Second *floating point* input parameters as 'F' and 'S' respectively, your entry would then be as follows:

```
PERCENT:(F,S)
```

Example 2: If you write a procedure called TEXT, which is to 'return' a *character string*, and you wished to pass in an *integer* parameter called 'W', the procedure name must end with **$**, and the input parameter must end with **%**:

```
TEXT$:(W%)
```

e) If you are not passing a value into the procedure *as a parameter* (there are alternative methods), simply press **EXE**. In any event, you can always add or change the parameter input information when EDITing your procedure. What you cannot do is change the procedure *name* - so be sure it is has the correct identifier if the procedure is to *return* a value or a string.

Entering the program from here on, editing it and saving it when you are ready, is identical to the EDIT option.

The EDIT option

You would choose this option whenever you wished to change or add to a program procedure that you have previously saved. You must, of course, know the name you gave the procedure and the location where it is saved. (See *The DIR option*).

a) Select EDIT from the PROG Menu.
The screen will display EDIT followed by a letter indicating the last location worked on.

b) Select the location of the program procedure you wish to edit (A:, B: or C:) by pressing the **MODE** key. (If Datapaks are not connected, **MODE** will have no effect).

c) Enter the name of the program procedure you wish to edit. If, for example, you wish to make some changes to a procedure called 'TEXT$', previously saved to RAM, the screen would look like this after your entry:

```
EDIT A:TEXTs
```

It doesn't matter whether you use capital or lower case letters.
You *don't* enter any 'bracket' information.
(If you wish to clear the program name to enter another, press **CLEAR/ON**: pressing **CLEAR/ON** a second time will cancel the edit operation and return you to the PROG Menu).

d) Press **EXE**.
The screen will display, on the top line, the name of your program procedure followed by a colon: if you are using the *parameter* method of passing values to the procedure, these will be shown, in brackets, after the colon just as you entered them.
The second line of the display will show the first line of your program.

You are now in a position to edit your program procedure. The use of the keys during editing is as follows.

UP This moves the position of the cursor to the *start* of the previous line.

DOWN This moves the position of the cursor to the start of the following line.

LEFT This moves the position of the cursor one character to the *left* – moving to the last character of the previous line when the beginning of a line is reached.

RIGHT This moves the position of the cursor one character to the *right* – moving to the beginning of the next line when the end of a line has been reached.

DEL This deletes the character to the *left* of the cursor.
If the cursor is positioned at the extreme left of a line, pressing **DEL** causes that line to be shifted up to join the end of the previous line. All the following lines are moved up.
If **DEL** and **SHIFT** are pressed together, the character *under* the cursor is deleted. If the cursor is at the extreme *right* of a line, the next line down is brought up so that the two lines become one, and all the ensuing lines are moved up.

EXE This key produces the effect of a 'carriage return' on a typewriter: it ends the entry on the current line and places the cursor at the beginning of the next line down. It is used to produce the following results.

Ending a program procedure line
Each complete program line should be terminated by pressing **EXE** when the cursor is *at its end*.

Breaking a line
If **EXE** is pressed while the cursor is in the middle of a line, a new line will be created below that line, and all other lines following it are moved down. The new line will contain all the characters that were under and to the right of the cursor on the original line.

Inserting a new line
If **EXE** is pressed while the cursor is at the end of a line, all following lines will be moved down and a space created for a new line to be entered: the cursor will be positioned at the start of this new line, ready for your entry.

CLEAR/ON
This key *clears* all the characters from the line on which the cursor is placed, and leaves the line blank ready for a new entry. If the line is already empty, it deletes the line.

To make an entry
Press the required character key, as normal: the entry will be made at the cursor position, and the cursor shifted right one place to mark the point for the next entry. Note that if the cursor is a flashing block, you will be entering letters, if it is an underline, you will be entering the characters printed *above* the keys.

When you have finished editing
Press **MODE**. The screen will then display a further Menu:

```
TRAN SAVE QUIT
```

You make your selection from this Menu in the same way as you do from other Menus. The options are:

TRAN This option TRANslates your program from the Organiser's programming language to a language that the machine uses. You must choose this option before you can run the program. A message is displayed on the screen telling you that translation is in progress.
If you have made a *syntax* error in your program (a spelling mistake in one of the language words, for example), it will be found during this translation process – which then stops, and the screen will display an appropriate message. Pressing **SPACE** returns you to

the program procedure in the EDIT mode, with the cursor positioned on the line where you have made a mistake, ready for you to make the correction.

When the translation process has finished, the screen will display the message SAVE, followed by a location. If Datapaks are connected, you can select these to save your written instructions (the *source* instructions) and the translated program (*object* code), by using the **MODE** key. *You are strongly advised to select* 'A:' *– to save your program in* RAM *– until it has been thoroughly tested and you are sure you will not want to change it.* (See Chapter 1.2).

Press **EXE** to save your program.

SAVE This option enables you to save a program without having it translated first. You may choose to do this if you have entered only part of the program procedure, and wish to continue at a later time. Programs saved without being translated first *cannot* be run.

Note: When saving an untranslated program, be sure you are saving it in RAM – 'A:'. You will lose memory space in your Datapaks very quickly if you don't.
A program procedure saved in RAM *overwrites* any existing procedure with the same name. A procedure saved to a Datapak does *not* overwrite any existing procedure of the same name: the existing procedure is 'locked up' so that it cannot be used again, and only the newly saved procedure can be accessed.

QUIT This option allows you to cancel your entire edit. It does *not* erase any previously saved versions of the program. You may choose to QUIT a program if you have merely examined it to see what it contains. When **Q** is selected, the screen displays the message:

```
QUIT
ARE YOU SURE Y/N
```

Pressing **N** or **CLEAR/ON** returns you to your program. Pressing **Y** returns you to the PROG Menu.

The RUN option

This option enables you to run a *translated* program either for actual use or for test purposes.

a) Press **R** to select RUN from the PROG Menu.
b) Use the **MODE** key, if necessary, to select the location of the saved program.

c) Enter the name of the program procedure you wish to run.

Note: 1) Your program procedure may call other procedures: this doesn't matter – only the location of the *named* program procedure is required.
2) If your program procedure uses the parameter (*brackets*) method of passing information to it, it *cannot be run* by using the RUN option. You must write a *test* program which calls the procedure, correctly passing the required values within brackets. Alternatively, you can return to the main Menu and enter the CALC mode to test the procedure. Thus, if your procedure is 'PERCENT:(F,S)', you would enter CALC and type in 'PERCENT:(?,?)' – with actual *numbers* in the brackets, separated by a comma.

d) Press **EXE**.
Your program will run, and when finished, you will be returned to the PROG Menu.

To *pause* the program while it is running, press **CLEAR/ON**: this 'freezes' the program. It can be re-started by pressing any other key.

To *quit* the program, press **CLEAR/ON** followed by **Q**. The screen will display the message 'ESCAPE', followed by the location and name of the procedure that has been interrupted. Pressing **CLEAR/ON** or **SPACE** returns you to the Menu.
This facility can be extremely useful to abort a program that is 'running wild' – *provided* you have not disabled the 'escape' facility within your program, and provided that the program is not waiting for a character INPUT.

During the *running* of your program, if there are any *programming* errors (rather than *syntax* errors, which are usually detected at the Transslation stage), Organiser will stop and report the error, an will tell you in which procedure the error occured. On pressin **SPACE**, if the *source* of the procedure is available (*you* instructions), the top line of the screen will display the location anc name of the offending program procedure, and the bottom line will display the message 'EDIT Y/N'.

If you choose to edit the procedure (press **Y**), you will be put in the EDIT mode, usually with the cursor flashing at the point in the program where the error was detected. This enables you to study the line and correct it (if indeed the error is there).

If you do not choose to edit the program at this time, press **N**.

The ERASE option

This option allows you to remove a program procedure from memory. If the procedure is in RAM, the space it occupied is made available for further use. If it is on a Datapak, the space it occupied is 'locked up' and cannot be used again.

a) Select ERASE from the PROG Menu.

b) Press **MODE**, if necessary, to select the location of the saved procedure.
c) Enter the name of the procedure you wish to erase.
d) To abort the erase operation at this stage, press **CLEAR/ON** twice. You will be returned to the PROG Menu.
e) Press **EXE**.
The second line of the screen will display the message 'ERASE Y/N'. Pressing **N** or **CLEAR/ON** returns you to the PROG Menu. Pressing **Y** erases both the translated version, if produced, *and* your written version of the named program, then returns you to the PROG Menu.

The DIR option

This option enables you to see a list (DIRectory) of all the program procedures saved to a particular location.

a) Select DIR from the PROG Menu.
b) Press **MODE** if necessary to select the location of the program procedures you wish to list.
c) Press **EXE** repeatedly to display each procedure name in turn. Pressing **EXE** when you reach the END OF PACK message will take you back to the beginning of the list.
d) Press **CLEAR/ON** at any time to return to the PROG Menu.

The COPY option

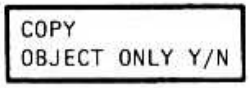
This option enables you to copy one or all of the program procedures in one location to another location. For example, you may choose to transfer from RAM to a Datapak a program that you have thoroughly tested, and know you will not wish to work on any further.

The destination of the program(s) to be copied must be different from the source: if you do not have a Datapak connected, you cannot make copies.

Note that copying to a Datapak places a greater drain on the battery supply: be sure your battery is fairly fresh before attempting to copy program procedures. Should you get the **LOW BATTERY** message, switch off immediately and replace the battery as soon as possible.

To make a copy:

a) Select the COPY option from the PROG Menu.
The screen will display



Organiser is asking you whether you wish to copy *just* the translated version of the program (the *object* code) or whether you want to save the translated version *and* your written program.
If you copy *only* the translated version *you will not be able to use the copied program for editing*. Thus, if you then go on to erase the program from its original location, *you will not be able to edit it again*.
If you are sure that your program needs no further work, you need copy only the translated version (the *object* code): this will require less than half of the memory space needed to save both the translated version and your own written version.
Press **Y** if you want to save the translated version only.
Press **N** if you want to save both versions.
Press **CLEAR/ON** *twice* if you wish to cancel the copy operation.

b) The screen will display 'FROM'.
Enter the current location of the program procedure(s) you wish to copy ('A:', 'B:' or 'C:').
c) Enter the name of the procedure you wish to copy, immediately after the colon, then press **EXE**.
If you wish to copy *all* procedures from the specified location, simply press **EXE**.
d) The screen will display 'TO' on the second line.
Enter the location you wish to copy the program procedure(s) to. ('A:', 'B:' or 'C:'). This must be different to the current location.
e) If you wish to rename your copied procedure, enter the new name. But be careful: if the procedure is 'called' by other procedures, its name will have to be changed in those too.
d) Press **EXE**.
The screen will display the message 'Copying...'. The time taken by the process depends on the amount to be copied. When copying is complete, you will be returned to the PROG Menu.

Note: If a procedure of the same name already exists at the destination location, that procedure will be deleted and the copied version will take its place. There cannot be more than one procedure with the same name at any location.

The LIST option

If you have a peripheral device such as a printer connected to Organiser via an RS232 Link, you can list a program procedure you have written on that device.

a) Select the LIST option from the PROG menu.
   The screen will display 'LIST A:' (or 'B:', or 'C:').
b) Use the **MODE** key to select the location of the saved program you wish to list.
c) Enter the name of the program procedure you wish to list.
d) Press **EXE**.

When listing is complete, you will be returned to the PROG Menu.

3.3

INTRODUCING THE PROGRAMMING LANGUAGE

The types of instruction

Organiser II's programing language 'OPL comprises a number of words or abbreviations. An example of one of the words is PRINT, and an example of an abbreviation is CLS – which stands for '**CL**ear the **S**creen'.

Each word or abbreviation tells the Organiser to perform a specific task, and in many instances, needs to be followed by *information* to enable the Organiser to perform that task.

PRINT, for example, tells the Organiser to 'print' on the screen the information that immediately follows it. The *instruction* comprises the word PRINT and the information that follows it. CLS on the other hand, is a complete instruction in itself – it needs no further information. It says to Organiser – clear the screen of whatever is currently displayed, and position the cursor at the top left hand corner of the screen.

Some of the words work in 'pairs' or groups. DO and UNTIL are an example of a pair of words. When Organiser sees the instruction DO, it expects to find, later on in the *same program segment or procedure*, the word UNTIL. If UNTIL is not found, it has problems. And so will you.

Some of the words need to be followed by information or *parameters* contained in brackets. If you have read Chapter 2.6 (*Using the Calculator*), you will have met a few of these already – SIN(), LOG(), and so on. These are *numeric* functions, because they work something out and give a *number* as a result.

There are also functions that work on *letters* or *characters*. They work something out and give a character or characters as a result. One example of this type of function is 'UPPER$()'. This function needs *text* or *characters* as the parameter in the brackets, and it instructs Organiser to – "look at the characters, and turn every lower case letter into a capital letter".

Did you spot that $ sign after the word UPPER?. It is not a mistake. It identifies the fact that UPPER is a language word that returns *characters* and not numbers. All of the OPL functions that return characters rather than numbers are identified this way - just as your own functions have to be so identified.

Why the $ sign? In computer terminology, a series of characters is called a *string*. One cannot use 'S' to identify a a function that works with a string – it could be confused with something else. So the $ sign is used. When you see this sign, don't say 'dollar', say 'string'. We'll be discussing strings in greater detail later on.

Thus, in OPL, there are words that are complete instructions in themselves (CLS), there are words that need information of some kind after them to complete the instruction (PRINT), and there are words that work in pairs or groups (DO and UNTIL). These types of word are called *Commands*.

There are also words that need *parameters* contained in brackets after them to complete the instruction (SIN() and UPPER$()): this type is known as a *function*. A function *returns* a value or a string (whereas a *Command* doesn't return anything). If a function returns characters rather than numbers, its *name* will end with a **$**, and it is known as a *string* function. Not all *functions* require parameters as part of the instruction: some, such as GET and DATIM$, get the information they need from other sources - such as the keyboard, or RAM.

The parameters that a function needs are also called its *arguments*.

Separating the instructions

Each OPL word and the information it may need associated with it is *one instruction* or *statement*. A program is a series of *statements*, written in sequence to produce the required result.

However, Organiser needs to know where one statement ends and another starts. For example, if you wrote 'PRINT *something* CLS' – meaning "display *something* on the screen and then clear the screen" (not a clever sequence if you want to read what was displayed!), Organiser would be confused. It would print '*something*', and then go on and try to print 'CLS', rather than clear the screen, as you wanted. It would see all the instructions as one *statement*.

Consequently, each statement *must* be separated from the next.

There are two ways of achieving this in Organiser II.

The first and simplest way is to ensure that each statement is on its own line. You achieve this by *pressing* **EXE** *after entering each complete statement* into Organiser. Pressing **EXE** acts like a carriage return on a typewriter. It finishes the line, and puts you down to the next line. More than this, it actually puts a special character at the end of the line (which you cannot see) – a character that Organiser recognises as marking the end of a statement.

So, using this method for the (silly) 'PRINT *something* CLS' example, you would enter each complete instruction or statement and press **EXE** so that it appears on its own line:

```
PRINT something
CLS
```

The second method is to enter a *space* folowed by a *colon* after the instruction, then enter the next instruction. Thus:


```
PRINT something :CLS
```

You can put (almost) as many instructions on one line as you like using this method – always finishing with an **EXE** keypress at the end. However, it makes your programs less easy to read when editing them (some of the instructions will 'vanish' off the screen, and you will have to keep using the cursor keys to scroll them back into view). Furthermore, your program instructions will take up slightly more room in Organiser II: instead of needing memory space for one **EXE** character, as in the first method, each instruction needs memory space for a *space* character and a *colon* character, to mark its end. The space required by the *translated* program is not affected.

Many programmers like to keep related statements on one line. Some have access to a printer to list their programs, so the number of statements one line doesn't matter – they will all be 'on view'.

But for those learning to program, the advice is – one statement per line. It is less likely to produce errors.

The way you end an instruction *must always* be one of these two methods. It is one of the 'rules' of the language. Like most rules, there is one exception - **EXE** need not be used to terminate the very *last* instruction of a procedure.

The unknown quantities – variables

When you write a program or procedure, you will undoubtedly want it to work on different sets of figures or characters each time you use it. You could re-enter the program to edit the values you wish to change – just as you can change the numbers when using the Calculator – but this, obviously, would be an extremely slow and laborious process. And quite unnecessary.

Organiser II allows you to set aside memory boxes to hold your 'unknown' quantities, and to give those boxes *names*. You then use those names to identify the unknown quanities.

The unknown quantities are called **variables**.

We have already seen variables 'in action' in Chapter 3.1, when discussing for example the 'percentages' calculation: we called the two numbers involved 'First Number' and 'Second Number'.

The names given to memory boxes that are going to store variables must obey certain rules. Organiser must know, for example, what *type* of information is being stored in the boxes. (This was covered in Chapter 1.2).

The *way* you tell Organiser to reserve memory boxes will be discussed later: here we shall concern ourselves with the different *types*, and how they are identified by their *name*.

The types of *data* information that can be stored are

Floating point numbers.
Integer numbers.

Character strings.

Each of these types is stored in a different way in the Organiser. The *identifier* you give to a variable name tells the Organiser what type of information it can expect to find in the associated memory boxes.

Floating point variables are those that have – or *can* have – a decimal point in them somewhere. The names for memory boxes set aside for floating point numbers have *no* identifier: the *lack* of an identifier in a variable name tells Organiser that the name refers to a floating point value.

When writing programs, actual numbers (rather than variables) *must* have a decimal point to identify them as a *floating point* type. Thus, **10.0** is seen to be a floating point number, but **10** is *not*.

Every floating point number or variable takes up eight memory boxes. Calculations made with floating point numbers are to 12-figure accuracy, and the numbers can (virtually) be of any magnitude.

Integer variables are those that can *never* have a decimal point in them. An integer *variable* is identified by a **%** symbol at the end of its name ('NUMBER%' for example).

When writing programs, an actual *number* is identified as an integer if it *doesn't* have a decimal point. Thus, in a program, '**10**' is seen to be an integer, and '**10.0**' is seen to be *floating point*.

If one integer number is divided by another, *only* the 'whole' part of the answer will be given – any fractional part is lost. In other words, the answer is also an integer.

On the other hand, if an *integer* number is used with a *floating point* number in a calculation such as multiplication or division – the answer produced will be a *floating point* number.

When programming, this is true whether the numbers are 'variables' referred to by their name, or actual numbers. For example, if a program includes an actual calculation such as **9/7**, the answer will be **1**: Organiser sees both of these numbers as *integers* and produces an answer that is an integer. To get the full floating point answer, *one* of them must have a decimal point in it – thus either **9./7** or **9/7.** would do.

(Note that when using the Calculator, *all* numbers, with or without decimal points, are treated as floating point numbers. Program procedures intended for use in the CALC mode must take this into account if they contain *integer* values).

Each integer number or variable requires two memory boxes only (compared with eight for floating point numbers). In addition to needing less storage space, calculations made using only integer numbers are considerably faster. However, in Organiser *Integer numbers can be in the range* **–32768** *to* **+32767** *only*. Values outside this range will produce an **INTEGER OVERFLOW** error message.

Character strings are a sequence of characters. The maximum length of a string *variable* – that is, the maximum number of characters the string variable can contain – is decided by you when you tell the Organiser to reserve space for the variable. The highest number you can specify is 255.

A *character string* variable is identified by a $ symbol at the end of its name. Thus 'TEXT$' is recognised by Organiser as a *string* variable.

A sequence of *actual* characters used in a program is identified by placing the sequence in *quotes*. Thus, the actual character string HELLO, if used after, say, a PRINT command, would be identified as a string as follows

```
PRINT "HELLO"
```

If quotes are *not* used for this string, Organiser would identify 'HELLO' as a *floating point* variable – and if you haven't reserved space for it, you would get an error. The maximum length of a string within quotes is 255 characters.

If the character string contains *numbers*, Organiser cannot use those numbers in a calculation: they are seen to be *characters* and not a *value*. OPL has provision for converting numbers in a character string to *actual* numbers, for calculation purposes – and for converting actual numbers to a string for combining perhaps with other characters.

Array variables

The variables discussed in the previous paragraphs were of the 'single' type – that is, each *name* related to one *item* of information. There are many instances, however, when you will want to have a *group* of closely related variables with the same name.

For example, you may have a series of costs related to one particular project. You could give the variable for each cost a different name – COST1, COST2, COST3, and so on. This could, however, make writing the program lengthy and tedious.

Organiser provides a solution for this type of problem by allowing you to have an *array* variable.

An array variable is one where each individual item of information is identified by an index number, enclosed in brackets *after* the main name for the variable.

Taking the 'cost' example again, the variable for each individual cost would be identified as COST(1), COST(2), COST(3) and so on. (How you tell Organiser to reserve space for an array variable will be discussed later – it is actually easier and shorter than telling it to reserve space for each variable as an individual item).

On the face of it, these array variables don't seem to be any advantage – they appear to be longer, for a start. However, the number in the brackets *can itself be a variable* – and this can make

for much shorter programs. This will become clear when we start preparing some sample programs. But to give you an idea now, let us suppose that it is necessary to make a fairly lengthy calculation on each of the cost items in turn.

If each item has its own name – COST1, COST2 and so on – the calculation would have to be made each time, with the appropriate variable name used in each calculation. Alternatively, the calculation could be a separate procedure – and it would then be necessary to pass the value of COST1, COST2 etc into that procedure. Both of these alternatives would entail repeating a series of instructions with only minor changes each time.

If an array is used, however, the calculation can be performed on the array, with an *integer* variable within the brackets as the index: the calculation would use something like COST(N%) as the variable on which it works. Now all that is needed is to repeatedly perform the calculation, *just changing the value of* 'N%' *each time*. This, as we shall see, can be accomplished very easily with just *one* set of instructions.

Array variables can be of exactly the same types as ordinary variables – floating point, integer and character string – and are identified in the same way. The only difference is they have brackets after them, enclosing the data that identifies their position in the array. Examples of array variables are

Floating point	COST(3)
Integer	CARD%(5)
String	MESSAGE$(7)

Thus 'COST(3)' is the third element in the floating point array COST(): the *total* number of elements in the array will have been determined by you when telling Organiser to reserve space for the array.

Choosing variable names

The rules governing the choice of the name you give to a variable are very simple:

a) They must contain no more than eight characters, *including* the identifier.
b) They must start with an alphabetic character.
c) They must contain only alphabetic characters and numbers.
d) They must end with the appropriate identifier – **%** for integer variables, **$** for character string variables.

These rules apply to both single variables and array variables: with array variables, the brackets and anything in them does *not* count towards the maximum of eight characters permitted in the name.

This allows you to choose names which will help you to identify the purpose of the variable when you are writing your program. For example, CHEQUE%, AMOUNT and NAME$ could identify three variables for holding a *cheque number* (which will never have a decimal point – so it can be an integer type), the *amount* a cheque is made out for (which could have a decimal point – for pence – and so must be a floating point type), and the *name* of the person it is made out to (which will be a character string).

You can, if you wish, use the *same* name for variables with different identifiers. In the above example, you may for instance choose to call them PAYOUT% (for cheque numbers), PAYOUT (for the amount) and PAYOUT$ (for the name). Organiser II will recognise these as *different* variables, because their identifiers are different.

When you become more experienced, you may choose to use shorter names – down to single or double letters: longer names obviously take up more space within Organiser. However, at first *clarity* and understanding of your program is more important, so choose names which suit your purposes best.

LOCALs, GLOBALs and parameters

When you write your programs, you must tell Organiser what space to reserve for the variables that you wish to use. You will recall, from Chapter 1.4, that each program *procedure* is brought into RAM for 'processing' *only as and when it is needed*. That is the time Organiser needs to know what space is required for the variables, or where to find them if space has been reserved elsewhere.

Organiser will reserve space for a variable in one of three different ways, depending on how the variable will be used. It is up to you, the programmer, to tell Organiser which way to use. The decision is yours entirely – Organiser obeys your instructions implicitly. But it expects you to follow the 'rules' of the way you select: if you don't, when you come to use a variable, Organiser may not be able to find it, and an error will occur when the program is used.

The three different ways are as **LOCAL** variables, as **GLOBAL** variables, and as *parameters*.

LOCAL variables are used *only* in the procedure in which they are named or 'declared'. They are declared at the beginning of the procedure, and space is reserved for them for as long as the procedure is in its 'running position' in RAM. When all the instructions in the procedure have been completed, the procedure is removed from RAM – and the space reserved for the **LOCAL** variables is cleared, ready for further use. If two or more procedures are in the 'running area' at the same time (because an instruction in one procedure tells Organiser to perform another procedure), each procedure has its own memory boxes reserved for

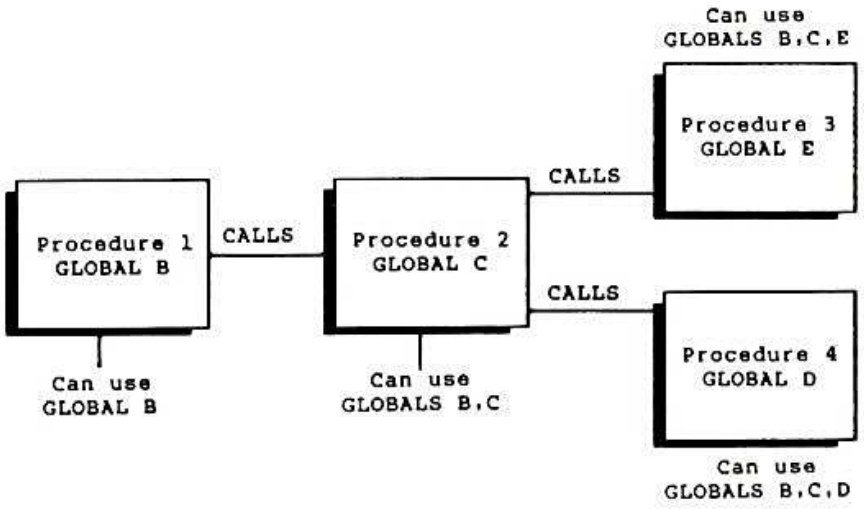


Fig. 3.3.1. The use of GLOBAL variables

its own **LOCAL** variables.

This means that if the same name is given to a **LOCAL** variable in two different procedures, then the variable will be quite distinct in each procedure. Thus, a **LOCAL** variable called COUNT% in one procedure would be quite different – and unaffected by – a **LOCAL** variable called COUNT% in another procedure. Each would have space reserved for it by Organiser, and the two would be treated as separate entities.

However, it *is* possible to pass the value held by a **LOCAL** variable on to another procedure as a *parameter*. For example, if **LOCAL** variables in one procedure are called 'WEE' and 'BIG', the values held by those variables can be used as parameters when calling the procedure named 'PERCENT:' (by entering them within the brackets – 'PERCENT:(WEE,BIG)'. We will discuss parameters later on.

GLOBAL variables can be used in the procedure in which they are declared, and in any procedures successively called by that procedure. A **GLOBAL** variable *cannot* be used in any procedure that comes *earlier* than the one in which it is declared. This is illustrated in Fig. 3.3.1 which shows a possible sequence of Procedures.

In this sequence, the **GLOBAL** variables B, C, D and E can be used in the Procedures as follows:

Variable B can be used in all Procedures.
Variable C can be used in Procedures 2, 3 and 4.
Variable D can be used *only* in Procedure 4.
Variable E can be used *only* in Procedure 3.

Variable C cannot be used in Procedure 1 because space will have been reserved for it *after* Procedure 1 has been put into the running area of RAM. Organiser looks *back* to find the variable, not forward: to find Variable C when obeying the instructions of Procedure 1, it would need to look forward to Procedure 2. (In actual fact, if Fig 3.3.1 represented the *entire* program structure, there would be no point in making Variables D and E 'GLOBAL', since each can be used only in the procedure in which it is declared.)

Just as with **LOCAL** variables, the information contained in **GLOBAL** variables can be passed on to other procedures as a parameter.

Whereas it is fine to use the same name for **LOCAL** variables in different procedures, it can be dangerous to repeat the name given to a **GLOBAL** variable in more than one procedure.

Parameters are the variables that appear in brackets immediately following the name of a function type of procedure.

When you write such a procedure, you give these variables names (e.g. 'F' and 'S' in PERCENT:(F,S)). When Organiser loads the procedure into the running area of RAM, it looks back at the 'calling' procedure to get the information to be used, and *copies* it into memory boxes reserved for the parameter names you chose.

Thus, when you want to use the PERCENT:(F,S) procedure from the Calculator (say), you would enter 'PERCENT:(35,90)' – or whatever numbers you choose – to 'call' that procedure. Organiser will place the procedure PERCENT:(F,S) into the running area of RAM, and will reserve memory boxes for the parameter names 'F' and 'S' for use in the procedure. It will copy into the memory boxes reserved for 'F' and 'S' the actual information '35' and '90'.

The memory boxes reserved for parameter names *within* a procedure cannot be used anywhere else except in that procedure – just as with **LOCAL** variables. The difference is, whereas the contents of memory boxes reserved for **LOCAL** variables can be changed, the contents of parameter memory boxes *cannot* be changed.

Normally you would use different variable names when *calling* the procedure – 'PERCENT:(WEE,BIG)', for example. In this instance, Organiser uses the memory boxes allocated to the names 'F' and 'S' to store the information it finds in the memory boxes allocated to 'WEE' and 'BIG'.

It is important to remember that this type of procedure can be run *only* by being called from other procedures (or from the Calculator, if it is a numeric function), and that the input para-

meters – whether values or variables – *cannot be changed* by the procedure.

The way Organiser II works when it reaches the name of a variable when running a program is this.

a) It first tries to find the variable's name among the *parameter names* (if any) and the **LOCAL** variables declared at the beginning of the current procedure.
b) If the variable's name is not found, it then searches among the **GLOBAL** variables declared in the current procedure.
c) If it still cannot find the named variable, it looks back at the **GLOBAL** variables declared in previous procedures still in RAM. Remember that, if one procedure 'calls' another, the 'calling procedure' will still be retained in the running area of RAM because it hasn't yet been completely processed. When a procedure has been completed , it is removed from the 'running area' of RAM to make room for another procedure. (See Chapter 1.4).

If Organiser still cannot find the named variable, it stops processing the program to announce **MISSING EXTERNAL** – followed by the name of the variable it couldn't find. Organiser assumes that the missing variable should have been declared by a previous procedure. This may not always be the case: the missing name may be that of a **LOCAL** variable that you mis-spelled, declared as a different type, or simply forgot to declare.

Non-declared variables

Before we leave the subject of variables, it should be mentioned that there are two types of variable that do not have to be declared as **GLOBAL** or **LOCAL**.

The first of these relates to the *memories* used by the Calculator – designated 'M0' to 'M9'. These are accessible to programs at any time, by simply writing their 'name' (e.g. 'M3'). Any information you store in a calculator memory stays in that memory – for use when in the CALC mode. Similarly, any information stored in such a memory whilst in the CALC mode is available for use in a program. These memories are of the *floating point* type.

The second type of variable – or more strictly, parameter – is used when creating, opening and using *files*. As you will see, when creating or opening files, part of the process entails defining or naming *fields*: the field names are used as variables controlling the input and output to the files, and are considered to be 'declared' when the file is created or opened. From that point, until the file is 'closed', these 'field name' variables behave as **GLOBAL**s. This will be discussed in greater depth in the file-handling Chapters.

When things go wrong

It would be as well to realise, at this point, that inevitably mistakes will happen when you are programming. The programmer who has *never* made a mistake doesn't exist.

Four kinds of thing can go wrong.

a) A command or instruction can be incorrectly written – spelled wrongly perhaps – or incorrectly used.
b) Information hasn't been passed correctly from one program procedure to another – variables are used incorrectly, or space has not been set aside correctly for your variables.
c) The *logic* of your program could be wrong: it doesn't perform the way you believe it should.
d) You make a spelling mistake in a message that is to be displayed on the screen.

Mistakes in category (a) will be discovered by Organiser when your program is TRANslated into the instructions that Organiser uses to run your program. In most instances, it will announce precisely the type of mistake you have made (NAME TOO LONG, BAD IDENTIFIER, SYNTAX ERR and so on), and, on pressing **SPACE**, will return you in the EDIT mode to your program listing, with the cursor at the point where the mistake was found. You can then examine your entry in the light of the error message supplied, and make a correction.

Mistakes in category (b) will usually be discovered when you *run* your program: the program will stop running, and the screen will display an error message related to the problem it encountered. It will tell you which procedure contains the error and, on pressing **SPACE**, will ask you if you wish to EDIT that procedure. If you press **Y**, you will be put in the EDIT mode, with the cursor flashing at the point in the procedure listing where Organiser could no longer obey your instructions because of the error. In most instances – but not always – this will be close to where an error has been made.

Mistakes in category (c) are more tricky ... the nightmare of all programmers. All you can do is check back over your flowcharts (provided you prepared them!), check back over your logic, and try to analyse, from the way the program is running, what is going wrong. Sometimes it helps to write extra *test* programs to see exactly what is occurring when your program is running.

The worst thing that can happen when a logic error occurs is that Organiser appears to have a mind of its own – wild things appearing on the screen, perhaps, or it may even seem to 'go dead'. This is the dreaded *crashing out* you may have heard about. In these circumstances,

Keep cool.
Press **CLEAR/ON**.
Press **Q** (for Quit).

This should cause the program to stop, and the message **ESCAPE** to be displayed followed by the procedure in which the break occurred. Pressing **CLEAR/ON** again will return you to a Menu, and you can start making your investigations. If you cannot stop the program running this way, you have but one recourse ... remove the battery and press **CLEAR/ON** for a second or two. You will completely reset Organiser, and lose *everything* in RAM (but not on Datapaks) in the process. Oh dear.

Note: 1) If you have used the `ESCAPE OFF` command in your program, you will *not* be able to stop it the way just described and you will definitely have to remove the battery. Take heed!
2) Programs can also be locked into 'infinite' loops by careless use of the OPL instructions `ONERR label` and `DO/UNTIL`, discussed in later Chapters.

Errors in the logic of a program can produce the most obscure results, often in a different part of the program to where the error has actually been made. Sometimes the error will occur only under certain conditions – when a variable has one specific value, for example. Locating such errors needs careful analysis of the program step by step. This is discussed in more detail, together with methods that can be used to avoid errors developing, in Chapter 3.17.

Errors in category (d) are simple. Enter the EDIT mode and correct your mistake.

3.4

FROM KEYBOARD TO SCREEN

OPL words covered
AT, CLS, GET, INPUT, PRINT, RETURN

Creating a Procedure

We are now in a position to start putting an actual procedure together. The first thing to be determined is the *type* of procedure it is going to be – and how information is to be passed into and out of it. A procedure, remember, can be a discrete part of a program or a complete program in itself.

Information can be passed *into* a procedure five different ways:

a) From the keyboard.
b) As **GLOBAL** variables, declared in a previous procedure.
c) As parameters, enclosed in brackets after the procedure name when it is called.
d) By accessing one of the CALCulator memories (floating point numbers only).
e) From a *file*.

Information can be passed *out* of a procedure six different ways:

a) Direct to the screen.
b) As a **GLOBAL** variable declared within the procedure, or in a previous procedure.
c) As a 'returned' value.
d) Through one of the CALCulator memories (floating point numbers only).
e) Into a *file*.
f) Through an external device such as a printer.

In this Chapter, we will deal with the first three ways to get information in and out of a procedure.

Naming the procedure

The name that you give to a procedure must obey certain rules – as detailed under the heading *The NEW option* in Chapter 3.2. The important points to remember are that the name must be no more than eight characters long – including any *identifier* for the type of parameter to be *returned* by the procedure – and that only letters or numbers can be used, starting with a letter.

To demonstrate the process we will create a program which will be a useful addition to your Organiser – to calculate any percentage of any number (e.g. 15% of 34.56). We will create this

program two different ways: first, as a *function* type of procedure – requiring the values to be passed to it as *parameters*, and giving the answer as a *returned* value. Secondly, we will develop the procedure into a complete program – which could be run from the main Menu. This will demonstrate both approaches.

For the first approach, we shall call the procedure PF, the 'F' indicating to us that it is to be a *function*, and to differentiate it from any name we give to our second approach: when naming procedures you can, of course, choose any name you wish – provided that it obeys the 'rules'. The name for this procedure has been kept short deliberately – to make it easier to use.

We will be 'returning' a *floating point* value from the procedure – so an identifier (**%** or **$**) is not needed at the end of the procedure name. (That's why the **%** sign isn't used in the name – it doesn't mean *percentage* to Organiser, it means *integer*).

The first step is to get into the programming mode – by selecting PROG from the main Menu. The PROGramming Menu will then be displayed, and since we wish to create a *new* procedure, we select the NEW option.

The screen will now display the message **NEW A:**. If another letter follows the word 'NEW', it means that Organiser plans to save the procedure, when finished, to one of the Datapaks: when initially writing and developing programs and procedures, it is advised that you save your work in RAM. So, if the screen is not displaying 'A:', press the **MODE** key until it does.

You must now enter the procedure name. Type in 'PF', so that the screen looks like this:

```
NEW A:PF
```

Now press **EXE**. The screen will display the name of the procedure, followed by a colon, and the cursor will be flashing in the space after the colon.

This is where, when using the *parameter* method of passing information to the procedure, we must 'declare' our variables. That is, tell Organiser to expect values to be provided as parameters when the procedure is used. We need two variables – for the *percentage* and for the *value*, and we shall call these 'P' and 'V' respectively. As *parameters*, they must be entered within brackets, so now enter '(P,V)' immediately after the colon, so that the screen display is

```
PF:(P,V)
```


Notice that, where more than one parameter is being declared, each must be separated from the one preceding it by a comma.

This is the first line of our procedure. We have named it, and we have declared our parameters. The line is complete: no more must be entered on this line, so press **EXE**.

The cursor will now jump down to the beginning of the second line. We have 'entered' the first line as an 'instruction': we have told Organiser to reserve space for the parameter variables 'P' and 'V'.

We must now tell Organiser to reserve space for any other variables we wish to use in this or subsequent procedures. We can write this particular procedure in a way that doesn't need any further variables – but first, we will use a variable to help with the calculation, and to demonstrate the process.

Declaring numeric variables

Since we are going to use the variable to 'assist' in making the necessary calculation in this procedure, and since we have chosen to *return* the answer, our variable will be a LOCAL one. We shall call it 'RESULT': as it will be used for a floating point number, it doesn't need an identifier.

We must now declare the variable – tell Organiser to reserve space for it. All variables – LOCAL or GLOBAL – must be declared at the beginning of a procedure, immediately after its name. To declare variables, you enter first the appropriate word LOCAL or GLOBAL and then the list, separated by commas. In this instance, we have but one variable (RESULT), and so we enter LOCAL RESULT. The screen will now look like this:

```
PF:(P,V)
LOCAL RESULT
```

Again, this line is now complete, and so you press **EXE**. The line 'LOCAL RESULT' will move up to the top line of the screen, and the cursor will be flashing at the beginning of the next line. If there were any GLOBAL variables to be declared, they would now be entered in a similar way – first the word GLOBAL, followed by the variable names separated by commas. It doesn't matter whether the LOCAL variables are declared before or after the GLOBAL variables, although it is probably better practice to declare the GLOBAL variables first.

Array variables are declared in the same way – but in this instance, it is necessary to tell organiser exactly how many 'elements' there are in the array, enclosed in brackets after the name. For example, if you decide to have a set of five variables called 'COST(1),

COST(2)-COST(5)', you declare the *array* as **COST(5)**. This tells Organiser that there are *five* variables with the name 'COST', and it will reserve space for them accordingly. It also knows that a specific variable will be identified by a number enclosed in brackets: thus, if you wanted to work on the third variable in the 'COST' array, you would name the variable as 'COST(3)'.

We are now ready to write the actual instructions to perform the calculation. You will recall that we are going to evaluate a percentage 'P' of a number or value 'V'. The formula is **V*P/100** – the Value multiplied by the Percentage as an actual number of hundredths.

Assigning values to numeric variables

To simply enter the calculation as it stands would not mean much to Organiser: it wouldn't know what to do with the answer. We must *tell* Organiser what to do – and in this instance, we want Organiser to store the answer in our variable called 'RESULT'.

This is a process called assigning a value to a variable: the value, in this instance, is obtained as the result a calculation. Let us take a look at how a value is assigned to a variable.

If you took algebra at school, the expression

X = X+1

would be absolutely meaningless: in algebraic terms, it is saying that 'a number is equal to itself plus one' – which is patently impossible. However, in *computer* languages, the = sign is used to mean *holds the result of* or *is to be*. It is an *assignment*.

Thus **X=X+1** says to Organiser: "I want you to store in the memory boxes allocated to the variable 'X', the result of what is already in there *plus one*. Organiser, on seeing this instruction, goes to the memory boxes allocated to the variable 'X', takes the value it finds there, adds one to it, and puts it back. The value has been increased by one.

We could write down

X = 42

In this instance, we are telling Organiser that we want the value '42' put into the memory boxes allocated to the variable 'X': it is a direct *assignment*. Whatever happened to be stored in the memory boxes before will be replaced by '42'.

When making numeric assignments, you must be very careful that you get the right *type* of number – floating point or integer. You will recall from the discussion on floating point and integer numbers in Chapter 3.3 that, in programs, Organiser treats any number without a decimal point as an integer. So if you wrote, for

example,

X = 5/2

the variable 'X' would be storing **2** as a result of the calculation, not **2.5** as you expected (or hoped). The reason is that Organiser sees the **5** and the **2** as *integer* values because they do not have a decimal point. Even though 'X' is a floating point variable, Organiser still divides one integer by another before it saves the result: integers divided by integers produce integers. To get the required result, you would have to include a decimal point after either one of the two numbers in the calculation. Thus, **X=5./2** or **X=5/.2** would result in a floating point number – **2.5**.

Similarly, if instead of 'X' the variable were called 'X%' – indicating that it is an integer variable, all that can be stored in the memory boxes allocated to it is an integer number. So

X% = 5/2

will *always* result in an answer of **2**, whether decimal points are included after the numbers **5** and **2** or not.

Finally, if using *variables* in the calculation, then the result depends on the variable *types*, rather than the values they contain. Thus

X%=F/S will always store the integer result.
X=F%/S will always store the floating point result.
X=F/S% will always store the floating point result.
X=F%/S% will store the *integer* result as *floating point*.

Organiser makes the calculation before making the assignment. So in the first case, although it makes the calculation as floating point, when it comes to make the assignment, it finds that the result must be stored as an *integer* – which doesn't allow for decimal points or fractional parts. In the last case, it makes the calculation on integer values – so any fractional part is ignored. It then finds it must save the *integer* result in a *floating point* variable – which it will do.

The reason for using integer values and integer variables, remember, is that they occupy only a quarter of the memory space floating point values need, and calculations with them are much faster (not that you'd notice the difference at the speeds Organiser works!). However, as we shall see, there are many occasions when *integer* numbers are more useful.

Returning now to our procedure, we have set aside the variable 'RESULT' to hold the answer to our calculatio:., so we now enter the line `RESULT=V*P/100`. This is a complete instruction to Organiser, and so we terminate it by pressing **EXE** again.

Our procedure is now complete except for one thing – we want to pass the answer, held in the variable 'RESULT', back so that it can be displayed (in the CALCulator mode, for example) or used by another procedure.

The RETURN command

The OPL word RETURN tells Organiser to leave the procedure, and go back to the previous procedure or, if none exists, to stop *running* the program and return to a Menu.

It is *not* a mandatory command: when Organiser reaches the end of the instructions in a procedure, it 'returns' anyway.

However, if we want to pass a value back, then RETURN must be used, followed by the particular value – or the variable holding the value.

To complete our 'PF' program, therefore, we need to enter one more line: RETURN RESULT.

As this is the *last* line of our procedure, there is no need to press **EXE**. No harm will be done if you do, however: all that will happen is that the cursor will be shifted to the start of the next line down.

The complete program should now look like this:

```
PF:(P,V)
LOCAL RESULT
RESULT=V*P/100
RETURN RESULT
```

Program 3.4.1. 'PF' Percentage function.

We will now put it to the test. First, we must translate it and save it: press **MODE**, and the screen will display the options TRANS SAVE QUIT. Select TRANS (press **T**), and after a moment, the screen will display SAVE A:. Make sure it does in fact say 'A:' (indicating the program will be saved to RAM), then press **EXE**. The procedure instructions that you have written *and* the translated version will be saved, and you will be returned to the PROG Menu.

As a *function* type of procedure requiring parameter inputs, it cannot be run directly from the PROG Menu: to do that, it must be 'called' from another procedure. Before we write that, however, let us see if our procedure works as a *function* in the CALC mode: press **CLEAR/ON** to return to the main Menu, then select CALC.

We are now going to 'call' our procedure. Enter after 'CALC:' **PF:(15,200)** (don't forget to use the **SHIFT** key for letters in this mode). Then press **EXE**.

You should get the answer **30**, indicating that 15% of 200 is 30.


```
PF:(15,200)
=30
```

(If you get the message **BAD IDENTIFIER**, check that you entered the colon after the procedure name – it tells Organiser that the function 'PF' is one of *yours*). Pressing **EXE** will return you to the calculation line – and you can now experiment with different percentages of various values by simply editing the numbers in the brackets. When you have had enough, press **EXE** once or twice to return to the main Menu, then select PROG again ... we are going to shorten our procedure!

From the PROGramming Menu, select EDIT: the screen should display EDIT A:. Enter 'PF' and press **EXE**. You will now be back in the EDIT mode, at the beginning of the procedure.

Earlier, it was mentioned that the OPL word RETURN is used to return a value to the 'calling' procedure. We used it to return the value contained in the variable 'RESULT'. We could, however, have used RETURN to return the result of a *calculation*. We will now edit our procedure to do just that.

Use the **DOWN** key to position the cursor at the start of the line LOCAL RESULT, then press **CLEAR/ON** to delete the line. This will leave a blank line, which we do not want. So press **DEL** (or **CLEAR/ON** again), and this will 'pull up' the next line and position the cursor at the end of the first line again. Use the **DOWN** key to re-position the cursor at the beginning of the next line – which reads RESULT=V*P/100.

We are going to change this line to read RETURN V*P/100, so first use the **SHIFT** and **DEL** keys to erase the characters RESULT= and then, with the cursor at the *beginning* of the line, type in RETURN *and a 'space'*. Do *not* press **EXE** – it is not necessary: the line has already been 'entered' – all we have done is to *change* it. What we must do now, however, is remove the superfluous RETURN instruction on the next line down. Press **DOWN** to position the cursor at the start of the next line – which should read RETURN RESULT, and press **CLEAR/ON** to delete the entire line. Your procedure should now look like this:

```
PF:(P,V)-
RETURN V*P/100
```

Program 3.4.2. 'PF' Percentage function, shortened version.

As you can see, we have no need for a LOCAL variable, and the entire procedure occupies no more than two lines. Notice too that, since there is no identifier at the end of the procedure name, the parameter returned will be a floating point type (which is what we

want). If the procedure name had ended with a % sign, then an *integer* value would be returned *irrespective* of whether the calculation produces an answer with a decimal point. In other words – don't try to use the % sign to indicate that the procedure is working out *percentages*!

Press **MODE** and translate and save the procedure as before, then test it to make sure it works (enter the CALCulator mode).

Calling one procedure from another

The procedure we have just written – which performs a numeric function – can be used only by being called from another procedure, or by being called from the Calculator. We have seen it work from the Calculator: now let us write a procedure to call it – a procedure that you can *install* on the main Menu, if you wish.

A procedure is 'called' by simply naming it, followed by a colon. This says to Organiser – "go and collect that procedure from wherever it is in memory and run it, then, when you've finished, come back and do the next instruction".

If the procedure being called requires input parameters, then these must be included in brackets, after the name and colon – the same way as you would call the procedure or function from the Calculator.

Enter the PROG mode, select NEW from the PROGramming Menu, and enter the following procedure exactly as it is written, using the process previously described (see also Chapter 3.2).

```
PC:
LOCAL PER,VA
CLS
PRINT "PERCENTAGE=";
INPUT PER
PRINT "OF VALUE=";
INPUT VA
CLS
AT 7,1
PRINT PF:(PER,VA);"%"
GET
```

Program 3.4.3. 'PC' Percentage program.

There are quite a few new words in this procedure: before we discuss them, you may like to translate and save it as before, and then test it by selecting RUN from the PROGramming Menu. The screen will display RUN A:, followed by the procedure name PC.

How did Organiser know that we wished to run that particular program? It is a pretty safe bet: we have just been writing and editing the procedure, and Organiser *assumed* that any further work will be undertaken on the same procedure. So it saves us the bother of having to type it in again. Of course, it isn't always right

– in which case, press **CLEAR/ON** to delete the entire procedure name, and enter the name of the procedure you do wish to run. Check too that the correct location is displayed: use the **MODE** key if the procedure you wish to run is on a Datapak.

Notice that, this time, you do not enter any parameters: the required values will be entered while the program is running.

Press **EXE**. The screen should display the message PERCENTAGE= on the top line. Type in the percentage you want – and note that the keyboard is already set for numeric inputs, and that the number you type appears immediately after the = sign. Press **EXE**, and on the second line the message OF VALUE= will be displayed. Type in the value you want the percentage of, and press **EXE**. The screen will immediately clear, and the answer, followed by a % sign (to indicate you have just calculated a percentage) will be displayed. To return to the Menu, press any key.

You can run this program from the Calculator – by entering *only* 'PC:' (don't forget the colon). You will be *prompted* for the numbers you have to enter – remember to press **EXE** after entering each one. Pressing any key after you have seen the result will cause the top line to display the procedure name, and the bottom line to display =0. Pressing **EXE** will return you for a repeated calculation.

To use the program from the main Menu, you will have to install it there first. This process is described in Chapter 2.2., under the headings *Customizing the main Menu – Restoring (or adding) an option*: note that when installing a procedure on this Menu, you enter *only* its name, *without* the colon. Having installed it, you can select it just like any other main Menu option. The way to remove it from the main Menu is also covered in Chapter 2.2.

The new OPL words used in this procedure will now be discussed.

CLS

This is a complete instruction, telling Organiser to clear the screen and to position the cursor at the top left corner.

PRINT

This command tells Organiser to print on the screen, starting from the current cursor position, the information immediately following it.

a) With one exception (see (b)), there must *always* be a *space* following the word 'PRINT'.

b) If the information is enclosed in quotation marks, it is treated as a *string* – and whatever is in the quotation marks is 'reproduced' on the screen. When a quotation mark follows the word PRINT,

there need *not* be a space (thus PRINT"GOOD" would be acceptable). However, it is probably better practice at first to always include a space after PRINT.

c) If the information is in the form of a variable, the value of that variable is printed. Thus, if a variable called COST has a value of 7.9, then PRINT COST results in **7.9** being displayed on the screen.

d) If the information is a calculation, then the calculation will be evaluated and the result printed on the screen. Thus, PRINT 4+5 results in **9** being displayed on the screen.

A number of items can be printed, provided they are separated by a semi-colon or a comma. With a comma, a space will be displayed between consecutive items. With the semi-colon, there will be no space between consecutive items.

Thus, PRINT "5+6=";5+6 would result in **5+6=11** being displayed - with no spaces between items.

PRINT "HELLO","WORLD" would display the two words separated by a space – **HELLO WORLD**. Using a semi-colon to separate the two words would result in **HELLOWORLD** being displayed.

If no information follows a semi-colon in a PRINT instruction, then *after* the instruction has been executed, the cursor is positioned at the next available space to the right on the screen.

If no information follows a comma in a PRINT instruction, then after the instruction has been executed, a space is left after the last character printed, and the cursor is positioned in the next space after that.

If neither a semi-colon or a comma is used after print information (or if PRINT is used on its own, without any information after it), the cursor is positioned at the extreme left of the next line down after the instruction has been executed.

INPUT

This word instructs Organiser to get an input of characters from the keyboard, and to display them on the screen as they are entered, finishing when **EXE** is pressed. The entered information is also stored for use in the program. If **CLEAR/ON** is pressed during the input of information, the information entered to that point is cleared and the entry can start again. The **LEFT** and **RIGHT** keys and the **DEL** key can be used to 'edit' the information during the entry process.

Organiser needs to know where to store the entered information, and so INPUT must *always* be followed by a

variable (INPUT NAME$, for example).

The variable must have been declared, either as a **LOCAL** in the same procedure, or as a **GLOBAL** in the same procedure or a 'calling' procedure. *You cannot use a variable that has been declared as a parameter.*

The *type* of variable determines the type of characters that will be input:

a) If the variable is a floating point or an integer (**%**) type, the keyboard will be set for numbers, and a numeric *value* will be entered. If a *character* is entered instead of a number, an error will occur.

b) If the variable is a string type (**$**), the keyboard will be set for characters: in this instance the number of characters that can be entered is limited to the number declared for the variable (see the next Chapter).

INPUT can also be used with program *files* – which are discussed in a later Chapter.

AT column%,line%

This OPL word needs two parameters (no brackets), separated by a comma. It positions the cursor on the screen at the specified column and line position, so that any display starts from that point. The column% value must be from 1 to 16 inclusive, and the line% value must be either 1 or 2 (for the top or bottom line respectively). The two parameters can be integer variables, actual values, or a calculation. If floating point values are used, Organiser will convert them to integer values.

GET

This word instructs Organiser to wait for *one* key to be pressed, and to return the ASCII value of that key. (The ASCII value is the *pattern* number – see Chapter 1.2, under the heading *Storing characters*).

Organiser needs to know where to store the ASCII value, and so the complete instruction *usually* takes the form VAR%=GET. Notice the use of an integer variable: ASCII numbers can never have a decimal point.

However, this word can also be used on its own, to simply hold up the processing of instructions until a key is pressed. This could be necessary, for example, after a PRINT instruction, otherwise immediately after the instruction has been obeyed processing will continue – and the display could be cleared before it has been seen!.

GET$

There is also a GET$ instruction, which returns the *character* associated with the pressed key. Like its int-

eger counterpart, the complete instruction is usually VAR$=GET$.

Let us now examine each line of this second procedure in turn.

PC:
The first line is simply the name of the procedure – but this time you'll notice, we have no parameters in brackets. This procedure is going to get its information from the keyboard.

LOCAL PER,VA
This instruction declares the floating point variables 'PER' and 'VA' – which are going to be used to store the numbers we wish to work on.

CLS
Clears the screen.

PRINT "PERCENTAGE=";
This displays the prompt "PERCENTAGE=" on the screen. The semi-colon means that the cursor will be positioned immediately after the = sign, ready for the entered value.

INPUT PER
This instructs Organiser to wait for numeric information to be input from the keyboard, accepting numbers until **EXE** is pressed. The numbers entered will be displayed on the screen, and stored on the **EXE** keypress in the floating point variable 'PER'.

PRINT "OF VALUE=";
INPUT VA
As above.

CLS
Clears the screen ready for the display of the answer. If this were not done, the answer would appear on the next line down, and the screen display would scroll upwards – which is messy.

AT 7,1
This places the cursor at the seventh character position along on the top line of the screen, ready for the display of information in the next instruction. This will set the answer roughly in the middle of the screen.

PRINT PF:(PER,VA);"%"
This instruction tells Organiser to display the result of calling the procedure 'PF:' using the variables 'PER' and 'VA' as

parameters. Note that, generally, these do not have to be the same names as the parameter variables used in the procedure 'PF:'.
The semi-colon tells Organiser to carry on with the PRINT instruction – without leaving a space – to display the next item of information. This is the '%' symbol, within quotation marks, and so it is displayed as it is.

GET
This instructs Organiser to wait for a key to be pressed, and since we have not told Organiser to store the information, it is effectively ignored.
This instruction is necessary at this point in the program to prevent Organiser returning immediately to a Menu (which it would do having finished processing our instructions), so that we have time to see the answer displayed on the screen.

We have now written two procedures to form one program. In fact, the procedure 'PF:(P,V)' was not absolutely necessary unless we wished to use it in other programs or as part of a Calculation. The line in procedure 'PC:', which reads

```
PRINT PF:(PER,VA);"%"
```

could have been written as

```
PRINT VA*PER/100;"%"
```

to produce the same result. Here, Organiser sees the instruction PRINT, and looks for the first item to display on the screen. It finds a calculation – which it performs so that it can display the result. It then looks on to see if that is the end of the instruction: it finds a semi-colon so, without leaving a space, it looks on to the next item to be displayed. This time it finds information within quotes – so it displays the information just as it is written.

These two procedures have, it is hoped, demonstrated a number of the principles involved in writing procedures. They provide additional facilities to your Organiser, for use either when making calculations, or to evaluate a particular percentage of a number.

For example, suppose you want to know what the price of an article would be with VAT added at 15%, if the article costs £34.56. You would select the CALCulator mode from the main Menu, and enter your calculation as:

34.56 + PF:(15,34.56)

Pressing **EXE** gives the answer – 39.744.

Warning: Don't try to evaluate the *original* price of an article that costs £39.744 including VAT at 15%, by *subtracting* 15% of

39.744 (a common mistake): you'll get the wrong answer. The formula for this type of calculation is

Original price = Cost*100/(100+percentage)

If you need to make such calculation, you can write another *function* type procedure. It would look like this:

```
OPF:(P,C)
RETURN C*100/(100+P)
```

Program 3.4.4. 'OPF' Original price function.

Here 'OPF', the name of the procedure, stands for 'Original Price Function', and the variables P and C stand for 'Percentage' and 'Cost' respectively. You could eliminate the percentage parameter 'P', and replace 'P' in the calculation by a fixed percentage value – 15, for example – but this would restrict the use of the procedure.

3.5
HANDLING CHARACTERS AND STRINGS

OPL words covered
CHR$, LEFT$, LEN, LOC, MID$, OFF, RIGHT$, RND

Declaring 'string' variables

In the last Chapter we dealt entirely with numeric variables. We saw that to 'declare' a numeric variable in a procedure, all that is required is to name the variable after the OPL word GLOBAL or LOCAL. Organiser knows exactly how much space to reserve for a numeric variable when it is declared. If it is an integer type Organiser will reserve two boxes to store whatever number the variable represents. If it is a floating point type, Organiser will reserve eight boxes.

This is not the case with *string* variables – identified by a $ sign after their name.

A string variable, remember, is one that contains *characters* (letters, numbers or symbols) – and each character needs one memory box. (See Chapter 1.2). The maximum number of characters allowed in a string is 255, and so Organiser *could* reserve 255 memory boxes for every string variable declared. However, this would be extremely space-consuming, since you will rarely need to have strings anywhere near 255 characters long. With only four string variables, for example, you would set aside over 1000 memory boxes – and probably actually use less than 100 of them.

Consequently, Organiser II allows *you* to decide how many memory boxes are reserved for each string variable, and this you do when you declare it.

When you decide to use a string variable, you must evaluate the maximum number of characters it is likely to represent. For example, if you are going to use a string variable to hold the name of a weekday, it need never be longer than the longest name – Wednesday – which has 9 characters. On the other hand, if you are going to use the variable to store an address, then you have to allow for what may be the longest address: usually around 50 characters.

Organiser will not let a string variable store more than the declared number of characters. It can't: it wouldn't know where to put them. If an attempt is made to store more than the specified number an error will occur and the program could stop running. (The clever people who developed Organiser built in a way to help you 'catch' such errors for action *within* your program, so that it needn't stop running. This is dealt with in a separate Chapter).

So, to declare a string variable you must include, in brackets after the variable name, the number of characters you want set aside in memory.

Thus, if you were declaring 'WEEKDAY$' and 'ADDRESS$' as LOCAL string variables at the beginning of a procedure, and you

decided that they need 9 and 50 memory boxes respectively, you would enter

LOCAL WEEKDAY$(9),ADDRESS$(50)

Any numeric variables would be declared on the same line, just as before, separated by a comma.

Array variables, you will recall, have a number in a bracket after them anyway: 'COST(5)', for example. How then do you declare a *string* array variable? In this instance, *two* numbers are included in the brackets, separated by a comma. The *first* number defines the number of *elements* in the array, and the *second* number specifies the maximum number of characters that any element in the array can hold.

Suppose, for example, we decide that we are going to have an array called 'WEEKDAY$()', each element of which is to hold the name of one of the days in the week. We will need *seven* elements in the array, and the maximum number of characters needed will be *nine*. Consequently, this array would be declared as a LOCAL variable (for example) as

LOCAL WEEKDAY$(7,9)

Here is a short procedure to demonstrate how arrays can be used. The procedure can be used to switch off Organiser (as an alternative to using OFF on the main Menu) so that, when it is subsequently switched on again, one of three messages will be displayed on the screen.

Even if you subsequently ERASE the procedure, it would be well worth entering in order to gain 'hands on' experience.

Enter the procedure as described in the previous Chapter. Briefly: select NEW from the PROGramming Menu, enter the name (GO – without a colon), press **EXE** *twice* (there are no parameter inputs), then enter the instructions, pressing **EXE** at the end of each line. (You can type in your own messages, within quotation marks, provided they are not more than 16 characters long – including the spaces).

```
GO:
LOCAL M$(3,16),N%
M$(1)="HI THERE..."
M$(2)="WELCOME BACK"
M$(3)="READY OH MASTER"
OFF
N%=1+RND*3
PRINT M$(N%)
GET
```

Program 3.5.1. 'GO' Switch off and 'welcome back'.

This procedure introduces two new words and some new techniques. Before we examine the new words and each line of the procedure to see what it is doing, press **MODE**, TRANslate and SAVE the procedure, then RUN it, from the PROGramming Menu. As soon as you press **EXE** to run the procedure, Organiser will switch off. Press **CLEAR/ON** to switch it back on again, and one of the three messages will be displayed. Press any key, and you will be returned to the Menu. If this program is installed (as 'GO') on the main Menu, you will be able to use it to switch off your Organiser by selecting GO instead of the OFF option – so that a message is displayed when you next switch on. Pressing any key will then display the main Menu.

Later on, we will use this technique to develop a 'Password' procedure which will allow you – and only you – to use your Organiser II.

Now, here are the details on the two new words used in this procedure.

OFF

This word, in a procedure, acts just like the OFF option on the main Menu. The difference is the procedure *is still in memory* – and Organiser is waiting to perform the next instruction following the OFF instruction. It patently cannot do this until it is switched on again, using the **CLEAR/ON** key. So instead of 'waking up' to display the main Menu, it continues with the procedure.

RND

This word is one of the numeric functions. It 'returns' a random number *between* (but not including) 0 and 1 – to 12 significant figures. Thus it will always give a decimal number between 0.000000000001 and 0.999999999999: this number can be used to provide a random number between *any* limits. Multiplying the random number by 5, for example, will make its range 0.000000000005 to 4.99999999995. If we take only the *integer* part and ignore everything after the decimal point, we get the range of random numbers 0 to 4. By adding 1, we get the range of random numbers 1 to 5. Thus **1+RND*5** will yield a random number between 1 and 5.

Now let us examine the procedure.

```
GO:
```

This is the name of the procedure: note that when you type in the procedure name after selecting NEW, you do *not* enter the colon – Organiser puts that in for you when you press **EXE** to 'enter' the line.

```
LOCAL M$(3,16),I%
```

There are no other procedures to this program, so all the variables are LOCAL. The M$(3,16) declares a string array variable that has three elements, each of which can be up to 16 characters long.

```
M$(1)="HI THERE..."
M$(2)="WELCOME BACK"
M$(3)="READY OH MASTER"
```

These three lines assign actual *strings* to the three string variable elements M$(1), M$(2) and M$(3).

```
OFF
```

When Organiser reaches this instruction during the running of the procedure, it will switch off. The program is still 'running' however – that is, it remains in memory. When Organiser is switched on again (**CLEAR/ON** key), it continues with the next instruction.

```
N%=1+RND*3
```

This assigns a random number to the integer variable N%. RND generates a number between 0 and 1 (but never including 0 or 1), which is multiplied by 3 and added to 1. The number generated will therefore lie between the values 1.0000000003 and 3.99999999997. This number is then assigned to an *integer* variable – and consequently the decimal part is lost completely: only a value from 1 to 3 is stored in N%.

```
PRINT M$(N%)
```

This demonstrates how a *variable* can be used to select an array element. The value of N% will be 1, 2 or 3. Whichever it is, Organiser will print the corresponding array element – M$(1), M$(2) or M$(3). Thus, one of the three messages will be displayed at random whenever the Organiser is switched on again – there is no knowing which of the three messages it will be. Fun, huh?

```
GET
```

You should be familiar with this instruction now. It tells Organiser to wait for a keypress (and so give you time to read the displayed message). When any key is pressed, the procedure is finished and Organiser returns immediately to the Menu. If you install this procedure on the main Menu, the main Menu will be displayed ready for you to select the application you wish to use on Organiser.

Note that this procedure can be shortened – to eliminate the need for the LOCAL variable 'N%', and the line 'N%=1+RND*3' – by re-writing the 'PRINT M$(N%)' line as 'PRINT M$(1+RND*3)'. The random number calculation will produce a floating point number,

but this will be converted to an integer by Organiser to determine which array element is required. We used 'N%' in the procedure to demonstrate how integer variables are defined alongside string variables, and to make the procedure easier to understand.

Joining strings together

There are a number of ways – including special OPL words – to manipulate strings of characters. You can join them together, to form a longer string, or take small sections out of them in various ways. We're going to look first at how you join strings together.

When dealing with numeric variables, to add two of them together you simply write, for example, 'VAR1+VAR2'. Organiser must be told what to do with the answer. If you wish to keep it for further use, you would assign it to another variable. Thus you could write 'VAR3=VAR1+VAR2' as a complete instruction.

The same applies to string variables. If the variable 'STRING1$' held the word 'LIGHT', and 'STRING2$' held the word 'HOUSE', then writing **STRING3$ = STRING1$ + STRING2$** would result in STRING3$ holding the word 'LIGHTHOUSE'. The two word strings have been added together (a process called *concatenating*) to form one longer string. Sufficient memory boxes must be available to 'STRING3$', of course, to store the longer string: if only five memory boxes are reserved when it is declared, then an error will occur, because the two strings together contain more than five characters.

The facility to add strings enables you to put together messages from various component parts. For example, you could have a string array 'WEEKDAY$(7,9)', each element of which holds the name of one of the days in the week – 'WEEKDAY$(1)' holding 'SUNDAY', 'WEEKDAY$(2) holding 'MONDAY', and so on. Another variable – say 'MESSAGE$' could hold the string 'TODAY IS '. (Notice the space after 'IS'). Adding together the strings MESSAGE$+WEEKDAY$(1) would produce the longer string 'TODAY IS SUNDAY'. Thus, an instruction such as

```
DAY$=MESSAGE$+WEEKDAY$(N%)
```

would save, in the string variable 'DAY$', the message 'TODAY IS' followed by the name of the weekday as determined by the value of N%. N% must be no greater than the maximum number of elements declared for the array 'WEEKDAY$', of course.

You can then write 'PRINT DAY$', to display the whole message. There are alternative ways to display the whole message:

```
PRINT MESSAGE$+WEEKDAY$(N%)
PRINT "TODAY IS",WEEKDAY$(N%)
PRINT "TODAY IS ";WEEKDAY$(N%)
```

The first of these tells Organiser to add the two strings together and to print the result – without using a further string variable to hold the result. The second and third methods use an actual string followed by a comma (which tells Organiser to leave a space after printing the first message) or a semi-colon (which tells Organiser *not* to leave a space – so it has to be included as part of the message) to separate it from the second string. In the last two instances the strings are not actually added together, but are displayed in sequence.

If you wanted the message "TODAY IS " to appear frequently throughout a program, then it is more economical on space to assign it to a string variable: it is then stored in memory only once, and can be called up whenever you wish to use it. If, however, "TODAY IS " appears only once in the program, then there is no point in assigning it to a string variable.

Any number of strings can be added (or *concatenated*) together, provided the result does not exceed either 255 characters or the maximum length declared for a string assigned to hold the result. You cannot add string variables to integer or floating point variables – these numeric variables must be converted to strings first. Similarly, if a string variable is holding a number (NUM$="1234", for example), Organiser cannot use that number to make calculations: the string variable must be converted to a numeric variable first. These 'conversions' are dealt with in a later Chapter.

Finally, you cannot *subtract* one string from another – 'STRING1$ = STRING2$ – STRING3$' would result in a syntax error. There are other ways of 'extracting' one string of characters from another.

Non-keyboard characters

Not all of the characters available in Organiser can be accessed direct from the keyboard. A form of £ sign, for example, is in the Organiser, available for display, but it is not obtainable by pressing any of the keys. Any of the characters not available from the keyboard can be selected by the OPL word **CHR$** – which must be followed by an integer number in brackets. This number relates to the ASCII number or pattern number for the desired character. The pattern number for the £ symbol for example, is 237, and so to display a £ sign, you would write

```
PRINT CHR$(237);
```

in a procedure. The semi-colon ensures that any value or variable to be printed follows the £ sign.

A full list of the characters and their pattern or ASCII numbers is given in the Appendix: note that there are no characters for pattern numbers from 8 to 31, or 128 to 159. These numbers are reserved for special actions by the Organiser: pattern number 16, for example,

produces a short 'beep' from the Organiser's sound system. You can create your own characters for pattern numbers from 0 to 7 through a procedure (given in Chapter 3.18): to display those patterns, you must use the 'CHR$()' instruction.

There may be occasions when you wish to display a quotation mark on the screen. Normally, quotation marks are used to define the start and end of a string – and the marks are ignored when the string is displayed. Thus `PRINT "HELLO"` will result in **HELLO** being displayed, without the quotation marks. You have two options. The pattern number for quotation marks is 34, so you could write

```
PRINT CHR$(34):"HELLO":CHR$(34)
```

Organiser gives you an easier way, however, and that is to write the quotation mark *twice* where you want it displayed. This is in addition to any quotation marks that define the start and end of a string. So to print **"HELLO"**, you would write

```
PRINT """HELLO"""
```

Similarly, to assign the string **"HELLO"** to a variable called 'STRING$', you would write

```
STRING$="""HELLO"""
```

Cutting the strings

Just as there are occasions when you will wish to join strings together, there will be times when you want only a part of a string. For example, one string variable may hold the character string

JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC

This is the first three letters of every month, in order. As we shall soon see, from such a string, you can 'pick out' the appropriate three letters as and when you want them.

The advantage of this is that *one* string variable is holding *all* of the month information – you don't need to have a separate string for every month name. However, to be useful, all the names must be of the same length.

Equally, you can 'pick out' a sequence of characters from the left or the right side of the string – you could pick out **LIGHT** and **HOUSE**, for example, from the string **LIGHTHOUSE**. Three OPL words allow you to do perform all these operations. These are now described:

LEFT$(string$,length%)

This OPL word needs two parameters: the name of the string variable or an actual string (string$), and the number of characters required from the left side of the string (length%). The 'length%' parameter can be an actual number, an integer variable, or a calculation producing an integer result.
It tells Organiser to take, from the specified string, the specified number of characters, starting from the left. Thus

LEFT$("LITMUSPAPER",3)

would tell Organser to take the three leftmost letters from **LITMUSPAPER** – in other words, **LIT**. Organiser also needs to be told what to do with these letters, to complete the instruction. Just as before, they can be assigned to another string variable, displayed, or added to other string variables. Here are three examples:

```
SHORTER$=LEFT$(string$,length%)
SHORTER$=LEFT$(string$,length%)+ANOTHER$
PRINT LEFT$(string$,length%)
```

RIGHT$(string$,length%)

This is similar to LEFT$, the difference being that Organiser takes the *rightmost* number of letters. Thus

RIGHT$("LITMUSPAPER",3)

would tell Organiser to take the letters **PER**.

MID$(string$,start%,length%)

This OPL word operates in the same way as the previous two. This time, however, *three* parameters are needed – the string, just as before, followed by the *start position* and the length required. The 'start position' tells Organiser where to begin taking the characters, the number taken being determined by the 'length%' parameter. Thus

MID$("LITMUSPAPER",7,3)

would tell Organiser to take three letters from **LITMUSPAPER**, starting from the seventh – in other words, to take the three letters **PAP**.

The following short procedure demonstrates how entering the number of a month can result in a display of the first three characters in that month's name: note the use of 'CHR$(63)' to produce a question mark after the word 'MONTH'.


```
MONTH:
LOCAL M$(36),N%
M$="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
CLS
PRINT "MONTH";CHR$(63)
INPUT N%
PRINT "That is",MID$(M$,(N%*3)-2,3)
GET
```

Program 3.5.2. 'MONTH' Month name from its number

Note the 'start point' calculation in the 'MID$'instruction – **(N%*3)–2**. (The brackets in this calculation are not really needed: they are included to help you understand it). When you enter a *month number* during the running of this procedure, that number must be related to the actual position of the month's name in the string M$. Each name takes up three characters, and so the month number, N%, is multiplied by three. However, this would put the start position for the first month at character 3 instead of 1, so we deduct 2 from the result. Now, whatever number N% is given (between 1 and 12), the correct start position for that month will be located in the string M$. Test it for yourself by choosing a month number, multiply it by 3 and deduct 2 from the result, then count that number of characters along the character string assigned to M$. You will get the first character of the name of the selected month.

When taking chunks out of a string using any of the three OPL words LEFT$, RIGHT$ and MID$, the original string is left intact. In the 'MONTH' procedure, for example, the string assigned to M$ is unaffected. However, just as with numeric variables, it is possible to store a new string back in a string variable used to make the selection. For example, if M$ has been assigned the character string 'LIGHTHOUSE' then the instruction

```
M$=RIGHT$(M$,5)
```

would result in M$ holding the character string 'HOUSE': the previous assignment is completely overwritten by the new characters, even though there are fewer of them.

You may wonder if that means 'blanks' are placed in all the remaining boxes. The answer is no: when Organiser saves characters in memory boxes allocated to a string variable, it also records the *number* of characters saved. It does this because you may not completely fill the number of boxes reserved – and it needs to know where to stop displaying characters when called upon. Thus, when it stores 'LIGHTHOUSE' in the string variable M$, it records the fact that there are ten characters in the string. When it stores 'HOUSE' it records the fact that there are five characters. Perhaps now you can guess why the maximum number of char-

acters there can be in a string is 255? It's the largest number a single memory box can contain (see Chapter 1.2).

How long is a string?

There are often occasions when you need to know how many characters are actually being stored in a string variable. For instance, suppose you wanted to display the contents of a string variable centrally on Organiser's screen, however many characters (up to 16) it contained. For this, you would need to know the number of characters in the string.

Just like the Greeks, OPL has a word for it ...

LEN(string$)

This OPL word 'returns' the number of characters in 'string$', which can be an actual string contained within quotations, or a string variable, or several string variables concatenated (added) together.

The following function-type procedure uses 'LEN(string$)' to display
a string centrally on Organiser's screen. It is a procedure that you may wish to use in various different programs, and so it uses a parameter type input: that way, you do not have to be concerned with what variable you use to pass the information into the procedure. The procedure also uses the OPL word AT to position the cursor.

```
CNTR:(M$,L%)
LOCAL W$(16)
W$=LEFT$(M$,16)
AT 1+(16-LEN(W$))/2,L%
PRINT W$
```

Program 3.5.3. 'CNTR' Function to centralise a display.

Notice that when declaring a string ('M$') as a *parameter*, it is not necessary to tell Organiser how much space to reserve for it. This is because the space is reserved automatically, according to the length of the actual string or string variable in the calling procedure – where memory space will already have been reserved for it.

Why have we introduced another string variable – W$ – into this procedure?. To avoid errors, we have to ensure that the first parameter in the AT instruction is not zero or less – which it could be if the input parameter M$ is longer than 16 characters. Organiser will not let you re-assign an *input* parameter, so we cannot have an instruction such as 'M$=LEFT$(M$,16)' in the procedure to ensure that M$ is never more than 16 characters long. One

solution to this is to introduce another string variable – 'W$', to 'pick off' the leftmost 16 characters of M$. If M$ is shorter than 16 characters, it doesn't matter: just as many characters as are available in M$ will be copied into W$. W$ is then used for the calculation of the start position for the subsequent display instruction.

To centralise the display of W$, we need an equal number of spaces each side of it. (Obvious?). To find the total number of spaces needed, we subtract the number of characters in W$ (as given by 'LEN(W$)') from 16 – the maximum that can be displayed on one line. This result is then divided by two, to get the number of spaces needed 'each side'. Adding one to this will give us the starting position for the display – the position at which the cursor is placed.

This procedure includes a 'line%' input parameter, so that when it is called, you can determine whether you want the information displayed centrally on the top line or the bottom line. Notice that the 'GET' instruction is not used in this procedure: that is left to the calling procedure to do, in case other instructions (perhaps to display on *both* lines) are required.

As we are not 'returning' a parameter, we do not have to worry about adding a $ symbol after the name 'CNTR'.

This procedure cannot be run in the Calculator mode. Consequently, we need another procedure to 'test' that it works. The following will demonstrate how to prepare such a procedure:

```
TEST1:
LOCAL W$(16),L%
PRINT "ENTER WORD(S)"
INPUT W$
PRINT "LINE 1 OR 2"
INPUT L%
CLS
CNTR:(W$,L%)
GET
```

Program 3.5.4. 'TEST1' Test CNTR procedure

Note that this procedure is not error-proof when running: you can enter a number greater than 2 for the line number variable, L%. This will cause an error in 'CNTR:', since '2' is the maximum that Organiser will accept for the line number in an AT instruction. While one tries to make procedures as fool-proof and as potentially error-free when running as possible, when they are used to *test* other procedures, the degree of error-prevention is up to the individual. As yet, we have not dealt with the OPL words that will allow us to check that numbers fall within specific limits: this is the subject of another Chapter.

Notice how the 'CNTR:' procedure is called by simply 'naming' it, with the appropriate parameters.

Once 'CNTR:' has been tested by running the 'TEST1' procedure, the 'TEST1' procedure can be erased. You may like to try the 'TEST' program *without* the 'CLS' instruction, entering '1' for the line number. In this instance, the word or words you enter will be centrally positioned – but any other characters on the selected line will remain on display if they are not overwritten. In other words, the 'AT' instruction doesn't clear the line – it merely positions the cursor in it. This can be useful sometimes to save reproducing an entire display when only one or two characters or numbers are being changed.

Where's that string?

In some types of program, it is useful to know whereabouts one character string occurs within another. Take for example the

JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC

string again. During the running of a program it may be necessary to know what *number* the month of "JUN" is. One could have an array of month names, and search the array for the element number corresponding to "JUN". However, as you have no doubt guessed, Organiser has an alternative method. The OPL word is LOC (short for LOCate).

LOC(long$,short$)

This instruction returns the position of the 'short$' string in the 'long$' string. As with other OPL words, Organiser needs to be told what to do with the answer – which will always be an integer value. As before, it can be assigned to an integer variable, displayed through a 'PRINT' instruction, or returned as a value. If the 'short$' string cannot be found in the 'long$' string, then the value returned is 'zero'.

Thus, a procedure to determine the month number, given the three-letter name of the month, could look like this:

```
MONTHNO%:(M$)
LOCAL Y$(36)
Y$="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
RETURN LOC(Y$,M$)/3+1
```

Program 3.5.5. 'MONTHNO%' Find month number from its name.

We want an integer number returned, so a % symbol is added to the procedure name: it then doesn't matter if the calculation in the 'RETURN' instruction line produces a floating point number. The 'LOC' instruction gives the position of the first character of the string M$ (our chosen month) within the longer string Y$ – i.e. how many characters along from the left the chosen month name starts.

Since each month name is three letters long, we divide this position number by 3. This gives an answer of one *less* than the actual month number, so to get the right month number, we add one. (Try it for 'FEB': the LOC instruction would yield '4', which divided by 3 gives 1.3333333333, and adding 1 gives 2.3333333333. The integer of this is '2', the actual month number).

The input parameter to this procedure must not be more than 3 characters long – otherwise an exact match will not be found. The procedure is not error free: if a match is not found, then the Month number returned will be '1'. That's because of the way the calculation is made: if there is no match, 'LOC(Y$,M$)' will give '0', which is then divided by 3 (still giving '0') and *'1' is added*. Hence the error. If this procedure were part of a serious program and not just for demonstration purposes, it would be necessary to prevent such an error occuring: as yet, we have not dealt with the words that allow us to do this.

The 'MONTHNO%:' procedure can be tested from the CALC mode, since a *value* rather than a string is being returned. For example, after 'CALC:' you would enter MONTHNO%:("JUN") (and
press **EXE**) to find the number for the month of June. Alternatively, you could write a short test procedure:

```
TEST2:
LOCAL M$(3)
PRINT "MONTH NAME"
INPUT M$
PRINT "Month No=";MONTHNO%:(M$)
GET
```

Program 3.5.6. 'TEST2' Test MONTHNO% procedure.

It would be well worth your while to enter the the MONTHNO% and the TEST2 procedures, in order to get the 'feel' of how Organiser handles searches. For example, when running TEST2, try entering just two letters for the search – Organiser will find the *first* match, and evaluate the month number accordingly.

If you do enter these procedures to gain experience in programming (how wise), you can easily erase them: they don't have to stay in your Organiser cluttering up valuable space.

3.6

DECISION MAKING

OPL words covered
IF/ELSEIF/ELSE/ENDIF, GOTO, AND, OR
Comparison operators >, <, =

Putting things to the test

In all the procedures that have been discussed so far, Organiser II has not been called upon to make any tests and act according to the result. Program 3.5.5 – to find the month number from three letters of its name – is an example where a test could eliminate the possibility of an error. You will recall from the description of this program that an error will occur if a match for the month name is not found, because '1' (indicating 'January') is added to the zero returned by the OPL word LOC.

What we want to say to Organiser in this program is "If you can't find a match for the letters that have been given, don't return a '1' – return a zero". To do this, the result of the LOC operation needs to be tested, and acted upon if it is '0'.

You may wish to write a program that finds the cheapest item from a whole range of items. To do this, you need to compare each with the next, discarding the most expensive one each time. To make the comparison, you need to *test* whether one value is greater than another – and act according to the result.

Similarly, you might want to have a program that sorts a series of names into alphabetical order. To do this, you need to test one name against the next, to see which should come first.

These are just a few of the many, many tests that need to be made in programs – tests which can result in a course of action.

Organiser II gives us all the ways we need to test the result of virtually everything it does. More than this, it gives us ways to act accordingly – *branching out* to take alternative actions if certain conditions are met, or not met, as the case may be.

We shall look first at the ways Organiser can perform tests.

The Test operators

Just as there are 'operators' which tell Organiser to perform mathematical tasks – such as + to add, – to subtract and so on, so there are operators which tell Organiser to make *comparisons*. These are <, >, and =.

That last one, the = sign, will undoubtedly cause some troubled frowns. Didn't we use = to make assignments – such as 'X=X+1',

and 'X%=42', and 'M$="WELCOME BACK"'?. We did indeed, but we are now going to use it in a different *context* – that is, in a different kind of instruction statement. The comparison operators are only *part* of the instruction.

These operators are used in the same way as mathematical operators: they separate two values (or two strings). They make statements about the two values, which Organiser is asked to check:

A<B	The value of variable A is less than the value of variable B.
A>B	The value of variable A is greater than the value of variable B.
A=B	The value of variable A is equal to the value of variable B.

Having made the comparison, Organiser needs a way to respond *that's true*. It does this by returning **–1**. If on the other hand Organiser finds the comparison statement to be *untrue*, it returns **0**.

We can put these comparisons to the test in the CALC mode. Select CALC from Organiser's main Menu, then enter **4<5** and press **EXE**:

```
CALC:4<5
--1
```

You have stated that '4' is less than '5', and asked Organiser to check it out. Organiser has responded with "*that's true*", indicated by '–1'. Press **CLEAR/ON**, then enter **4>5** and press **EXE**. This time Organiser will respond with '0' – indicating that the comparison is *not true*: '4' is not greater than '5'. Try other numeric values for yourself, using each of the comparison operators in turn. Every time, Organiser will respond with '–1' if the comparison statement is true, or '0' if it is untrue.

Having satisfied yourself that Organiser II gets it right every time where numbers are concerned, enter **"A"<"B"** and press **EXE**.

This time you have said that the letter 'A' is *less than* the letter 'B' or, to express it differently, *comes before* the letter 'B'. Organiser agrees with the statement.

Now try **"ABC"<"ABD"**. That, too, Organiser will agree with. But Organiser will not agree with **"ABC"<"AB"**: there are only two characters in 'AB', and so 'AB' is less than 'ABC'. Finally (unless you wish to continue experimenting), try **"AAA"<"AB"**. Organiser will agree, because even though there are three characters in

"AAA", the first *two* are 'less than' the first (and only) two characters "AB": if you were preparing an alphabetic listing, that's the way you would want it to be.

Thus, as far as strings are concerned, Organiser is capable of detecting whether they are the same, or if not, which should come first *alphabetically*, no matter how many characters (up to the maximum of 255) the strings contain.

Actually what happens is Organiser ccompares the *pattern numbers* or ASCII values of each pair of characters in turn.

The comparison operators can also be used in pairs, to make other statements:

>=	Means 'equal to or greater than' or '*not* less than'.
<=	Means 'equal to or less than' or '*not* greater than'.
<>	Means 'greater than or less than' or '*not* equal to'.

Thus, **3<>4** means "'3' is either greater than or less than '4'" or "'3' is *not* equal to '4'". Note the statement can be expressed positively or negatively with equal validity.

The comparisons work on numeric or string variables, as well as on actual numbers: with variables, Organiser checks the contents or value of the variables of course – not the variable *names*.

However, it is important to note that, while comparisons can be made between integer and floating point numbers or variables (e.g. **COUNT<MAXIMUM%**), comparisons *cannot* be made between numeric values and strings. (Thus **COUNT<TOTAL$** would produce an error). Even if a string is, in fact, holding a number, it must be converted to a numeric variable first: conversions from one type of variable to another are dealt with in a separate Chapter.

Having made the comparison and produced a result of '–1' or '0', Organiser can be told what to do by using the `IF ELSEIF ELSE ENDIF` set of OPL words. But before we come to these, here is a trick used by experienced programmers to save a lot of programming space and often greatly simplify program structures – although the 'trick' *appears* to be more complicated.

First, are you aware that '–1' multiplied by '–1' gives a result of '+1'? If not, test it on the Calculator. If you are happy with the fact, look at the following expression:

–1*(4=4)

Looks very complicated, doesn't it – certainly like nothing you were taught at school. Let us make some sense of it. Like Organiser, we'll first examine the part in brackets. This is saying '4 is equal to 4'. Organiser will check this out – and agree, which it signifies by a '–1'. So the expression now becomes:

–1*(–1)

The result of this calculation is +1. If in the brackets we had written **(4=5)**, then Organiser would not have agreed with this and the expression would have become:

–1*(0)

The result this time would be '0': any number multiplied by zero is zero. Now, suppose instead of actual numbers in the comparison we used a variable or a function, this expression is saying, in plain English *"If the test in the brackets is true, give a result of '+1'. Otherwise, if it is not true, give a result of '0'"*.

What use is this? Well, now have another look at the last line of Program 3.5.5:

```
RETURN LOC(Y$,M$)/3+1
```

Remember it was stated that if a match is not found, an error will occur, because of that '+1' at the end. Now have a look at this alternative:

```
RETURN (LOC(Y$,M$)/3)-1*(LOC(Y$,M$)<>0)
```

If you take it piece by piece, it is not really as bad as it looks. In fact all we have changed is the last part of the instruction. If `LOC(Y$,M$)` is *not* equal to '0', it means a match has been found for the string M$ in the string Y$ – and so `(LOC(Y$,M$)<>0)`, as a true statement, resolves itself to **(–1)**. That multiplied by the '–1' gives '+1' – and the correct month number will be returned from the procedure. If, on the other hand, a match is not found then `(LOC(Y$,M$)<>0)` is *not* a true statement, and the end of the instruction becomes **–1*(0)** – which is zero. Adding zero to the zero resulting from the first part of the instruction – `LOC(Y$,M$)/3` – means zero is RETURNed, indicating "no month". Just what we wanted.

If you have saved the procedure "MONTHNO%", edit the last line so that it is as shown above, then run the 'TEST2' procedure (Program 3.5.6) and choose letters you know will not match. Now, instead of getting a '1' to indicate (incorrectly) 'January', you will get '0'.

What has been demonstrated here is a technique: it has been used to add or not add '1' according to the result of a test. Any number could have been added, by simply replacing the '–1' with the appropriate '–' number. Similarly, any number could have been *subtracted* by replacing the '–1' with a '+' and the number. For example, to subtract 5 if a comparison is true, you would have '+5*(comparison)'.

As mentioned before, this technique is used frequently by

experienced programmers but rarely by novices. That is because the alternative, which we are now about to discuss, is easier to understand since it more closely follows the language we speak.

IF a comparison is true...

There is a group of words in OPL that allow you to tell Organiser to test comparisons and take actions according to the results. They are IF, ELSEIF, ELSE, and ENDIF.

Of this group, if you use IF, you must *always* use ENDIF. You cannot have one without the other. When you use IF, you can if the procedure requires it also use ELSEIF as many times as you wish, and/or ELSE once, before the ENDIF instruction.

IF

This tells Organiser to check out the comparison that immediately follows it. If the comparison is true, Organiser will obey the *next* instruction(s), until it comes to an ELSEIF, or an ELSE, or an ENDIF. Then, if it hasn't reached an ENDIF, it *jumps* all the intermediate instructions and goes straight to the ENDIF.

If the comparison is not true, Organiser will look through the following instructions until it comes to an ELSEIF or an ELSE, or an ENDIF. It will miss out the instructions immediately following the 'IF comparison' instruction.

ELSEIF

This is similar to IF – it needs to be followed by a comparison. The result is treated the same way as the result of an 'IF comparison'. If the comparison is true, it carries on with the next instruction(s). If it is not true, it searches for the next ELSEIF, ELSE or ENDIF.

ELSE

This says to Organiser – "the previous IF and possibly ELSEIF comparison instructions have all been untrue, so what I want you to do is this ...". Thus Organiser will perform the instructions following the ELSE instruction if all previous IF and ELSEIF comparison instructions in the group proved to be untrue.

Let us look at an example, using actual values rather than variables, so that you can see what is happening.

```
IFTEST:
IF 4=4
  PRINT "FOUR=FOUR"
ELSEIF 4=6
  PRINT "FOUR=SIX"
```



```
ELSEIF 5=5
  PRINT "FIVE=FIVE"
ELSE PRINT "THE COMPARISONS
PRINT "FAILED"
ENDIF
```

First of all, notice all the indents. *You don't have to enter the spaces when typing such a procedure into your Organiser.* They are there to help you see what is happening: this will be even more useful later on, when we *nest* IF instructions. You can enter the spaces, if you wish, when typing in your programs, to make them easier to understand – and indeed, many programmers do. It can actually help to eliminate errors of ommission.

Now, let us examine this little part of a program. The first instruction says to Organiser, "If the comparison '4 is equal to 4' is true ...". Before Organiser goes any further, it checks out the comparison. Lo! It *is true.* Organiser then looks to the *next* instruction, to see what it should do. That says PRINT "FOUR=FOUR". So it displays "FOUR=FOUR" on the screen. Now what? It looks to the *next* instruction ... and that is an ELSEIF. Thus, it has completed all of the instructions it must do 'IF 4=4'. So now it looks for the ENDIF instruction, and continues obeying instructions from that point on – missing out completely all the intermediate instructions without even looking at them.

If the first line had read IF 4<>4, then things would have been very different indeed. This time, Organiser would not agree that 4<>4 ('4 is not equal to 4'), and would look through the next instructions until it came to an ELSEIF, an ELSE or an ENDIF. In this example, it would come to ELSEIF 4=6. It wouldn't agree with that either. So the search continues for the next OPL word in the series ... and so it comes to the ELSEIF 5=5 comparison instruction. This it will agree with – and so it will carry on with the next instruction, PRINT "FIVE=FIVE". Then it reaches the ELSE – so it has done all it has to do *if* the comparison is true, and it jumps to the ENDIF instruction.

If the ELSEIF 5=5 instruction had read ELSEIF 5>5, Organiser would not have agreed ('5 is greater than 5?'– not true), and so the search would have continued. In our example, it comes to the ELSE instruction. This is what it must do if all else fails – which this time, will be the case. So it obeys the next instructions to display a message on the screen. It then reaches the ENDIF and all of this part of the program is over ... it continues obeying the instructions following ENDIF. ENDIF is a complete instruction, requiring one line to itself (or which must be followed by a space and a colon, if more than one instruction is being entered on the line).

As mentioned earlier, the only two words of this group that *must* go together are IF and ENDIF. The other two words ELSEIF and ELSE are optional: you use them according to program require-

ments. However, ELSE must be the *last* of the words used before an ENDIF, and it can be used only once in a series.

Now let us take another look at Program 3.5.5. We can use the 'IF' construction to make the test of whether LOC(Y$,M$) is zero or not, and tell Organiser to act accordingly. Thus, *instead* of the existing last line, we could have the following:

```
IF LOC(Y$,M$)=0
  RETURN 0
ELSE RETURN LOC(Y$,M$)/3+1
ENDIF
```

That certainly looks easier to understand than the other 'last line' we developed earlier on, doesn't it? However, as you can see, it is a longer program.

We can shorten it slightly. We have already seen that when Organiser makes a test on a comparison, it responds with '–1' for true, and '0' for not true. When it looks to see the *result* of a comparison after an IF instruction, *it takes* **0** *to be untrue, and* ***any other value*** *to be true – not just* **–1**. Thus, when LOC(Y$,M$) returns a '0', Organiser will regard that as being *untrue*, while any match would be regarded as *true*. Consequently we could have written the last few lines this way:

```
IF LOC(Y$,M$)
  RETURN LOC(Y$,M$)/3+1
ELSE RETURN 0
ENDIF
```

As you can see, the *order* of things has changed: when a match is found for M$ in Y$, Organiser regards the result as being *true* – so it will continue with the next instruction. If a match is not found ('0'), it will regard that as being *untrue*, and so will return a zero.

Finally, the ELSE instruction can be dropped completely from the procedure, by writing these lines as follows:

```
IF LOC(Y$,M$)
  RETURN LOC(Y$,M$)/3+1
ENDIF
RETURN 0
```

This time, if the IF instruction is *true*, Organiser returns the month number just as before. If it is not true – there is no match for M$ – then Organiser goes to ENDIF, and continues the instructions. The next instruction tells it to RETURN 0.

We have now discussed a number of different ways to write one small part of a program. They vary in length and understandability. They will also vary in speed of execution – but the difference is so small you'd never know it. However, if you were dealing with a sequence which is repeated thousands of times in a 'loop', the differences in time taken would become more significant.

Remember in Chapter 3.1 it was stated that there are many ways to write a program? You will now understand why.

IF – IF – IF instructions

Sometimes one comparison test is not enough to determine a course of action to be taken. For example, the pattern or ASCII numbers for the characters '0' to '9' are 48 to 57 respectively: you may wish to test that a character lies within this range. Your instructions to Organiser would be something like: "IF the character pattern number is *greater* than 47 and IF it is also *less than* 58, then it is a number character, and here's what I want you to do ...".

In this example, *two* tests need to be made. First it is necessary to check that the character pattern number is *greater* than 47: if it isn't, then it is not one that we require, so any further testing is unnecessary. If it *is* greater than '47' – that is, equal to a value of 48 or higher – it is then necessary to check that it is less than 58. If this is also the case, then the character number lies within the required range of 48 to 57. If the character pattern number is equal to 58 or more, then it is outside the required range, and not wanted.

Organiser allows us to tackle this problem two different ways. The first way uses the IF instruction discussed in previous paragraphs, so we will examine this first.

IF instructions can be 'nested'. That means within *one* set of IF instructions, you can include another set. The following solution to the above problem will demonstrate: in this, the character pattern number is represented by the variable 'CPN%'.

```
IF CPN%>47
  IF CPN%<58
    PRINT "CPN% IS A NUMBER"
  ELSE PRINT "CPN% IS OVER 57"
  ENDIF
ELSE PRINT "CPN% IS UNDER 48"
ENDIF
```

The first line says to Organiser "IF it is true that the value of CPN% is greater than 47, obey the next instruction. If not, go and find an 'ELSEIF', 'ELSE' or 'ENDIF' *at this level* of instruction." So far Organiser has encountered only one IF, and so it is at the *first* level. If the statement is *true*, Organiser moves on the the next instruction. That says to Organiser "IF it is true that CPN% is less than 58, obey the next instruction". This is the second time Org-

aniser has encountered an IF instruction, so it is now at the *second* level.

Let us assume, for a moment, that Organiser has reached this point in the program (because it is true that CPN% is greater than 47), and CPN% is indeed less than 58. It obeys the next instruction – which is to display the message "CPN% IS A NUMBER" on the screen – and then finds, *at this level*, the next instruction after that starts with an ELSE. It has completed the instructions at this level, and looks for the ENDIF – saying to itself "I've done everything required at this level". It will also have done everything required *at the first level* – having obeyed one set of instructions, and so will look for the ENDIF marking the end of the IF sequence at the first level. The procedure is completed.

If Organiser had reached the line IF CPN%<58 and found that to be *untrue*, it would look for the ELSEIF, ELSE or ENDIF instruction *at this level* – and so reach the line ELSE PRINT "CPN% IS OVER 57". This it will do, then see that the next instruction is an ENDIF – for this level – and will jump to the ENDIF at the first level, as before.

Now let us assume that Organiser found the comparison in the first line to be *untrue*: CPN% is not greater than 47, because it is either equal to 47 or less than 47. It ignores all the rest of the instructions until it finds an ELSEIF, ELSE or ENDIF at the first level – which it does at the line ELSE PRINT "CPN% IS UNDER 48". It obeys that instruction (there is not a comparison test to make), then finds an ENDIF, so all is complete.

You will now appreciate how indenting the IF statements and their ensuing set of instructions helps to understand at which level Organiser is operating. *Every sequence must be complete, finishing with an* ENDIF. You cannot have a sequence such as

```
IF (comparison)
  IF (comparison)
    PRINT "something"
ENDIF
```

The ENDIF in this sequence will relate to the *second level*, and the first level has not been properly terminated. An error would occur. Every *level* must be properly terminated with an ENDIF instruction.

As mentioned earlier, this process is called *nesting*. You can have up to eight nested sequences in Organiser – but this includes other instructions which can also be nested. We'll be dealing with these other instructions in a separate Chapter.

Using 'AND' and 'OR' to link comparisons

OPL has two words – AND, and OR – which are called **logical operators**. They are used to combine two *numeric* values in a special way – which depends on whether the numeric values are floating point or integer types. We can use the logical operators

AND and OR to link comparisons. There follows a description of how the operators work. This is given for completeness, and so that you may, at a later time, understand how to use them for more sophisticated applications. It is *not* necessary to understand how they work when using them to combine or link comparison tests, and so you may choose to skip the next few paragraphs.

—oOo—

Floating point numbers can be linked by AND and OR to produce the following results ('A' and 'B' represent floating point numbers).

A AND B Produces –1 if both A and B are non-zero. (Note: On some versions of Organiser II, the test is made only on the least two significant *digits* in the *binary* values of A and B. These must be non-zero for a –1 result). Otherwise the result is zero.

A OR B Produces –1 if *either* A *or* B is non-zero. Otherwise, if *both* are zero, the result is zero.

When used with integer values, these two words operate on the *bits* in the binary equivalent of the numbers. You will recall, from Chapter 1.2 under the heading *Computers count differently*, we discussed how eight switches can represent any number from 0 to 255, and that each 'switch' is, in effect, a binary digit. The switches represented the values 128, 64, 32, 16, 8, 4, 2 and 1 if 'on', and '0' if 'off'.

The AND operator compares each 'switch' or binary digit for the two numbers. If the same switch in both numbers is 'on', the corresponding switch in the 'result' is also switched on. If *either* is switched off, then the corresponding switch in the result is also switched off. To demonstrate this, let us take two integer values, 23 and 14, and examine how they are represented in binary digit form, and the result of **23 AND 14**

Binary digit or 'switch value'	128	64	32	16	8	4	2	1
'23' represented in binary	0	0	0	1	0	1	1	1
'14' represented in binary	0	0	0	0	1	1	1	0
Result of 'ANDing' the two	0	0	0	0	0	1	1	0

As you can see, where both binary digits are '1' there is a '1' in the answer. Where either or both is '0', there is a '0' in the answer. The decimal answer is found by adding the values for the '1' binary digits – i.e. 4 + 2, which is 6. Thus, **23 AND 14** is equal to 6. You can test this out for yourself in the CALC mode, by entering:

INT(23) AND INT(14)

and pressing **EXE**. It is necessary to convert the values to *integers*, because in the CALC mode, Organiser uses floating point numbers, whether they are entered with a decimal point or not.

The value of making such combinations will become apparent to you only when making sophisticated computations. The process is used, for example, to *mask* out unwanted numbers. (Organiser uses this system to convert lower case letters to upper case letters when you install a program onto the main Menu).

The action of the OR operator is similar: in this instance, the result of ORing two binary digits will be a '1' if *either* or *both* of them is a '1'. **23 OR 14**, for example, would result in the binary digits representing the values 16, 8, 4, 2 and 1 being 'on' –

giving a decimal value of 31 (16 + 8 + 4 + 2 + 1). Again, you can put this to the test in the CALC mode by entering **INT(23) OR INT(14)**. Experiment with other values – both ANDing and ORing - and see if you can deduce which 'switches' or binary digits are being tested.

—oOo—

The result of a comparison, remember, will always be –1 if it is true, or **0** if it is untrue. AND and OR can be used to check the result of two (or more) tests. If *all* the tests linked by AND are true, the result is **–1**. Thus

(True comparison)AND (True comparison) AND (True comparison)

would result in '–1', but if any one or more of the comparisons were *not* true, the result would be '0'.

When linked by OR, *any* of the comparisons being true – whether it is one or more –. would result in **–1**. If *all* the comparisons are *untrue*, the result is **0**.

The AND and OR words can be used in combinations, but it is important that you enclose in brackets the parts you want treated together. For example,

((Comparison A) AND (Comparison B)) OR (Comparison C)

would result in '–1' if either

a) *both* Comparison A and Comparison B are true.
or b) Comparison C is true.

In this next example,

(Comparison A) AND ((Comparison B) OR (Comparison C))

Comparison A *must* be true, and *either* Comparison B or Comparison C – or both – must be true for the result to be '–1'.

How does all this help with our IF instructions? It means that two or more comparisons can be combined to produce a result that the IF instruction can work on. Remember, Organiser looks to see if the result of any checks it has to make after IF is zero or *a value*.

Thus, an instruction line such as IF (A=B) AND (C>D) would cause Organiser to obey the next instruction if it finds that A is equal to B *and* C is greater than D. If either of these two comparisons is not true, then Organiser searches for the next ELSEIF, ELSE or ENDIF instruction.

If we look again at the problem of determining whether an ASCII

number is for a numeric character (ASCII numbers 48 to 57), this can be written as

```
IF (CPN%>47) AND (CPN%<58)
  PRINT "CPN% IS A NUMBER"
ENDIF
```

Notice that, this time, it is not possible to determine from the IF instruction whether 'CPN%' is greater than 58 or less than 47: if this is a program requirement, then the extra IF instruction will be required, as given in the earlier program example. The first line of this program segment says, in 'English': "IF the value of CPN% is greater than 47 *and* is also less than 58, (i.e. – if it is within the range 48 to 57 inclusive), obey the next instruction".

A 'Password' Procedure

Here is a short procedure that will enable you to prevent other people from using your Organiser II – to protect confidential information stored in it, for example. It operates in a very similar manner to Program 3.5.1. When run, you will be asked to enter a 'Password' of no more than 16 characters (which you must be very careful to remember!). Organiser will then switch itself off. When you (or anyone else) switches the Organiser on, it will be necessary to enter the Password. If it is entered incorrectly, Organiser will again switch itself off. Once tested, you can install the procedure on the main Menu: to use it you would select 'PASSWORD' instead of OFF to switch off, then enter your password.

```
PASSWORD:
LOCAL P$(16),I$(16)
PRINT "PASSWORD";CHR$(63)
INPUT P$
CLS
OUT::
OFF
PRINT "ENTER PASSWORD"
INPUT I$
CLS
IF I$<>P$
  GOTO OUT::
ENDIF
```

Program 3.6.1 'PASSWORD' Security switch-off

Note that this procedure uses a *label*, ('OUT::'): labels are discussed in the next Chapter. Be sure to enter the label correctly – that is, with two colons after it – in both places in the procedure.

It must be stressed again that you must remember your Password - otherwise it won't only be other people that cannot use your Organiser!

3.7

CREATING OPTIONS AND JUMPING AROUND

OPL words covered
MENU, GOTO, ABS, INT, Labels

Offering a Menu

In Chapter 3.1 we discussed a program to calculate the rolls of paper or emulsion required to decorate a room. This program could now be written using only the OPL words that have been discussed so far. However, with just one or two more words, the program can be made easier to use – requiring less effort to enter the necessary information.

You will recall that this program requires answers to various questions (Fig. 3.1.4). Are you going to use paper or emulsion as the decorating material? Do you wish to decorate the ceiling or walls? Do you wish to make another calculation on the same room, a new room, or have you made all the calculations you want?

All of these questions could have been answered by using the 'INPUT' instruction, but this would mean carefully analysing the information entered at the keyboard. Not impossible, of course. It just so happens there is a much easier way – and that is the way that Organiser uses itself when offering you options. The Menu.

Organiser allows you to create your own Menus, and to act according to the option selected. The OPL word, not suprisingly, is 'MENU'.

The complete instruction looks like this:

integer% = MENU(options$)

'Option$' can be a string variable, or an actual string. Unless you are going to use the set of options several times in different places in the program, it is probably best to use an actual string – that is, a series of characters enclosed in quotes.

This string lists all the options, each separated from the next by a comma. Our decorating program requires three sets of choices – between paper and emulsion, ceiling or walls, and same room, new room or finish. The three 'strings' we will use, therefore, will be:

"PAPER,EMULSION"
"CEILING,WALLS"
"SAME,NEW,END"

Now, how does this instruction work? Well, when Organiser meets a MENU instruction, it displays the *string* on the screen, in capital letters (even if you entered lower case letters), with spaces instead of the commas. Organiser sorts out whether it can get two or more words on a line without splitting them up – you don't have to worry about that. It may be, of course, that you have more words in your option list than can be diplayed on the screen at a time. If this is the case, then the words 'off the screen' can be brought into view by using the cursor keys – in just the same way as the words on the main Menu. In other words, your Menu will appear in the same fashion as any of Organiser's own Menus.

Selection of the required option is made in the same way too – by pressing the key for the initial letter of the required option, or by using the cursor keys to position the cursor over the required option, then pressing **EXE**. It obviously makes sense to try to keep the initial letters of the options different – so that pressing the appropriate key immediately selects the option.

When an option is selected, Organiser looks along the list to see which one it is. Organiser then returns a number related to the position of the selected option in the list. Thus, if the 'NEW' option were to be selected from the "SAME, NEW, END" Menu, Organiser would return the value '2' – indicating that the *second* word in the Menu string has been selected.

Organiser needs to know what to do with this number – hence the complete instruction is

integer% = MENU(option$)

It places the number for the selected option in the variable 'integer%'. You can, of course, give this variable any name you choose. But note it need only be an integer variable: decimal points will never be involved.

You will now have a variable that you can examine and act upon – using the IF instruction, for example.

If **CLEAR/ON** is pressed while one of your own Menus is displayed on the screen, then the value 'returned' will be '0'. You can use this to stop the program running and return to the main Menu. Or you can ignore it in your program. Any other key has no effect: the Menu remains displayed until an appropriate input has been made.

Jumping away

There are numerous occasions in programming when, having performed one or more tasks, you will then want to jump away to another part of the program. Our 'Decorating Materials' program has at least one example of this. At the end, when all the calculations have been made, we are offering options to make another calculation for the same room, or to start again with another room, or to finish altogether. If, say, another calculation for the same

room is required, we need a way to get back to the relevant part of the program – the choice between 'Paper' or 'Emulsion' (see Fig. 3.1.4) – so that room dimensions don't have to be re-entered.

Organiser provides the solution for us, with the OPL word 'GOTO', and by the use of **labels**.

Let us look first at 'labels'. A label is simply a marker in the program. It has a name – that we give it – and it is identified as a *label* to Organiser by following the name with *two* colons. The label acts as one complete instruction. It says to Organiser "Mark this place in the program: I shall be using it from time to time". The use of two colons is important: remember that Organiser identifies *one* colon after a name as an instruction to 'call' or run the procedure with that name.

To jump from any point in the *same* procedure to the label, you simply use the GOTO instruction, followed by the label name *and the two colons*. Note that you cannot jump to a labelled point in any other procedure – only to a labelled point in the *same* procedure. This means you could use the same label name in several procedures, should you so wish. But for clarity, you would probably not want to do so.

Thus, part of a procedure might look like this:

```
instructions
ENCORE::
instructions
GOTO ENCORE::
```

The 'jump' can be made either *back* to an earlier part of the procedure, or *forward* to a later part of the procedure. If jumping *back*, it is important that you have a way *out* of the procedure in the intermediate instructions – otherwise the program would never end! You will be in a continuous loop, and the only way out then is to press **CLEAR/ON** followed by **Q** – which breaks into the prcedure and quits it completely. This is not a satisfactory way to exit a procedure – it is an 'escape route' should things go wrong.

Before we learn any more words in OPL (and there are still plenty to come), let us put those we know into practice, and write the 'Decorating Materials' program.

The Decorating Materials program

We have already defined what we want this program to do, and we have prepared a flowchart of the way we are going to approach the problem (Fig. 3.1.4).

Let us now dot a few i's and cross a few t's.

We know that we will want to jump back from the last Menu to either the very start (to calculate for a new room) or to the point where a choice is made between Paper or Emulsion. We also know, now, that jumps like this can be made only in the same procedure.

So our program down to at least the 'Paper/Emulsion' decision, and the Menu offering the choice of further calculations or finishing, must be in the *same* procedure. It would make sense, too, to keep the display of the answer in the same procedure.

You will notice that there are two 'boxes' in the flowchart where a decision has to be made between 'Ceiling' and 'Wall'. There is no point in writing this part twice – so we will make it into a separate procedure, and 'call' it when it is required.

You will also notice that there is a box to calculate the emulsion required for the walls, and a box to calculate the emulsion required for the ceiling. The actual *calculation* in each instance is the same: area divided by coverage. The areas are different, of course, but if we used a *function* type of procedure, passing the area in as one of the parameters, then one procedure would cover both of these calculations.

The same is true with the calculations for the number of rolls of paper required: we have already ascertained that the rolls required for the walls can be calculated by

Rolls = (Room perimeter/1.75) / integer(33/room height).

For the ceiling, the number of pieces that can be cut from one roll is given by the integer of 'the roll length divided by the room *width* (or its *length*)'. The number of strips required altogether is the room *length* (or *width*) divided by the width of the roll. The number of *rolls* required is the total number of strips needed, divided by the number of pieces we can get from a roll.

We have set values for the length and width of a roll, in feet, as 33 and 1.75 respectively. So the calculation of the number of rolls required for the ceiling becomes

Rolls for ceiling = (room length/1.75) / integer(33/room width)

You will see the similarity between the two 'paper required' calculations: providing we pass in the correct parameters, we can use one procedure to cover both.

That leaves us with just one item not covered – getting the coverage of the emulsion. As this is used only once, we may as well include it in the main 'start and finish' procedure.

We thus need to prepare four procedures: the main 'control' procedure, the 'Ceiling or Wall?' procedure, the calculation of rolls required, and the calculation of emulsion required. These will contain the instructions as follows (compare the procedures with the flowchart of Fig. 3.1.4):

Main procedure

```
DECOR:
NEW::
Get room measurements.
```

Calculate room perimeter and areas.
MORE::
Get the option – Paper or Emulsion.
IF Emulsion, get coverage of emulsion.
'Call' the 'Ceiling or Wall' procedure.
IF Ceiling, 'call' Calculate Emulsion, for ceiling.
ELSE 'call' Calculate Emulsion, for wall area.
ENDIF
ELSE 'Call' the 'Ceiling or Wall' procedure.
IF Ceiling, 'call' Calculate Paper, for ceiling.
ELSE 'call' Calculate Paper, for walls.
ENDIF
ENDIF
Display the answer with appropriate message.
Get the option – Same, New, or End.
IF Same, jump back to MORE::
ELSEIF New, jump back to NEW::
ELSE End.
ENDIF

Note how we have already started paving the way for actually writing the program by including some of the IF instructions, in broad terms. Note too how the *labels* have already been positioned.

Ceiling or Wall
COW%:
Get – and return – the selected option (as a number).
Calculate Emulsion
CALEM%:
Calculate and return *gallons/pints* required, (area/coverage), using input parameters for the area, and coverage per *gallon*.

Calculate Paper
CALPAP%:
Calculate and return the number of rolls required, using input parameters: we want the answer to the nearest *next* whole number.

Before we actually start *entering* our programs procedures, we should consider the variables required. We also need to have some messages 'up our sleeve' when we display the answer. We have two choices of material, and two choices of 'part of the room': to demonstrate the use of array variables, and the use of *adding* strings, we will put the messages into two arrays, and combine them as needed.

How about the variables – for room height, width, length, area, and so on: should they be LOCAL or GLOBAL? We are going to pass parameters into and return values from of *all* of the 'sub' procedures – and so we need use only LOCAL variables in each

procedure.

As you can see, we have pretty well buttoned up the entire program requirements before we start to write the actual instructions. All we need to do, now, is dream up names for all the variables and so on. We can do that as we go along.

The 'Ceiling Or Wall' Procedure – which we have called 'COW%' – is going to be a simple MENU instruction, returning the integer number relating to the choice. It is because we are returning an *integer* value that the **%** symbol is included after the name.

```
COW%:
RETURN MENU("CEILING,WALLS")
```

Program 3.7.1 'COW%' Ceiling or Walls?

You can test this procedure by entering the CALCulator mode, and typing in **COW%:** (don't forget the colon). The screen will clear and the two words 'CEILING' and 'WALLS' will be displayed. Use whatever method you like to select one of them – and you will be returned to the CALC mode, with the lower line displaying a number corresponding to your choice: '1' for 'CEILING', and '2' for 'WALLS' (provided you entered them in that order in the MENU instruction!). Note what happens if you press any other key – and what happens if you press **CLEAR/ON**. When you are satisfied that everything works, return to the PROGramming Menu, to enter the next procedure.

The 'Calculation of Emulsion' Procedure – which we have called 'CALEM%' – needs some discussion. First, note that the **%** symbol is used after the procedure name – to indicate that we wish to return an integer value: we don't really want to know we need a fraction of a pint, since we would have to buy another *whole* pint if a fraction is involved. What we *do* need, however, is a way to round *up* any fractional part of the answer.

Organiser has a function word INT, which gives the integer value of any number contained in brackets after it. This number is always rounded *down*: thus, **INT(8.3)** gives a result of 8. This is not what we want. However, *negative* numbers are also rounded *down* – thus **INT(–8.3)** gives a result of –9. (You can test these out in the CALCulator mode, if you wish).

Making the number *negative* gives us the *value* we want – but we don't want it to be negative: Organiser has the answer with another OPL word – ABS (short for ABSolute value). This function takes the number in brackets, and whether it is positive or negative, it will always return the number as a positive value. Thus **ABS(–8.3)** will give 8.3.

By using the *two* words, and by making the answer *negative*, we can now force Organiser to give us an integer value, rounded *up*.

Thus, let us say the calculation of pints required produced an answer of '8.3'. We can get the answer we want – '9' – by making this value negative, then taking the *absolute* value of the *integer*. Thus **ABS(INT(–8.3))** gives **9**. (Test it in the CALCulator mode).

The other point to be made before writing this procedure is that we have decided the input parameter for the coverage of the emulsion will be square feet per *gallon*. We will need to convert this to pints (by dividing the coverage per gallon by eight).

So that you can understand the operation of this procedure, we will write it first using a LOCAL variable to help with the calculation. Then we will shorten it to a two-line procedure that doesn't need the intermediate variable.

```
CALEM%:(AREA,COVER)
LOCAL V
V=COVER/8
V=AREA/V
RETURN ABS(INT(-V))
```

Program 3.7.2 'CALEM%' Calculation of Emulsion, long version

(Remember that you enter the *name* first, press **EXE**, then enter the input parameter names in brackets: the input parameter names do not form part of the procedure name).

The variable 'V' is first used to hold the coverage of a *pint* of the emulsion – 'COVER/8' – and is then used for the calculation of number of pints used – 'AREA/V'. Note the complete instruction says to Organiser "Take the value stored in memory boxes called 'V', divide it into the value stored in memory boxes called 'AREA', and put the answer back into memory boxes 'V'". The last instruction we have discussed – it rounds *up* the answer to the next whole number.

The entire calculation can be achieved in the last line, thus:

```
CALEM%:(AREA,COVER)
RETURN ABS(INT(-AREA/(COVER/8)))
```

Program 3.7.3 'CALEM%' Calculation of Emulsion, short version.

Make sure you get the correct number of brackets at the end! If you are ever in doubt, the total number of *right* brackets should always be the same as the total number of *left* brackets.

Again, this procedure can be tested in the CALCulator mode, by entering **CALEM%:(value,value)** – putting in your own values. If all is well, you will find that you will always get a rounded up whole number, indicating the number of *pints* required. Remember that the second parameter represents the coverage *per gallon*).

The 'Calculation of Paper' Procedure – which we have called 'CALPAP%' – is very similar to the previous procedure, in that we

wish to return the next whole number *up*, so that we get the required number of rolls. Depending on whether the calculation is for the walls or the ceiling, the input parameters will be 'room perimeter or length', and 'room height or width'. So we shall call them 'POL' and 'HOW'. We will miss out writing a long version of this procedure, and go straight into the *two-line* version:

```
CALPAP%:(POL,HOW)
RETURN ABS(INT(-(POL/1.75)/INT(33/HOW)))
```

Program 3.7.4 'CALPAP%' Calculation of Paper

Make sure you enter the brackets correctly in this procedure. As before, you can test that the procedure works without errors from the CALCulator mode, using your own values for the input parameters. You can, of course, check that it gives the right number of rolls, by using parameters for a room you have already decorated (if you can remember how many rolls of paper you used!). Remember that this calculation just takes room width and length, and ignores alcoves, windows and doors: *usually*, these tend to balance each other out.

The main Decorating Program Procedure – which we have called 'DECOR' – can now be written. We are going to need variables for room length, width, and height, to hold the room perimeter, the wall and ceiling areas, to hold coverage per gallon of emulsion, to hold the answers to our Menu selections and the answer to the resulting calculation, and to hold our 'messages'. We will also have a variable to hold the 'Question mark' character.

We *could* cut down on the number of variables used, but for understandability, we'll allow each value to have its own variable. When entering the following procedure, you do *not* have to indent any lines: they are indented here to make the procedure easier to understand. All other spaces, punctuation and so on should be observed, however, in order to get a reasonable display on the screen.

```
DECOR:
LOCAL WIDTH,HEIGHT,LENGTH,PERIM,WALL,CEIL,COVER,PE%,
      CW%,ANS%,CW$(2,7),PE$(2,8),Q$(1)
CW$(1)="CEILING"
CW$(2)="WALL"
PE$(1)="PAPER"
PE$(2)="EMULSION"
Q$=CHR$(63)
NEW::
CLS
PRINT "WIDTH-IN FEET";Q$
INPUT WIDTH
```

```
CLS
PRINT "LENGTH-IN FEET";Q$
INPUT LENGTH
CLS
PRINT "HEIGHT-INCHES";Q$
INPUT HEIGHT
CLS
HEIGHT=HEIGHT/12
PERIM=2*(LENGTH+WIDTH)
WALL=PERIM*HEIGHT
CEIL=LENGTH*WIDTH
MORE::
PE%=MENU("PAPER,EMULSION")
IF PE%=0
  RETURN
ELSEIF PE%=2
  PRINT "SQ.FT PER GAL";Q$
  INPUT COVER
  CW%=COW%:
  IF CW%=1
    ANS%=CALEM%:(CEIL,COVER)
  ELSE ANS%=CALEM%:(WALL,COVER)
  ENDIF
ELSE CW%=COW%:
  IF CW%=0
    RETURN
  ELSEIF CW%=1
    ANS%=CALPAP%:(LENGTH,WIDTH)
  ELSE ANS%=CALPAP%:(PERIM,HEIGHT)
  ENDIF
ENDIF
CLS
PRINT CW$(CW%),PE$(PE%);
AT 1,2
IF PE%=1
  PRINT "ROLLS = ";ANS%
ELSE PRINT ANS%/8;"GALS + ";ANS%-(8*(ANS%/8));"PINTS"
ENDIF
GET
ANS%=MENU("SAME,NEW,END")
IF ANS%=1
  GOTO MORE::
ELSEIF ANS%=2
  GOTO NEW::
ENDIF
```

Program 3.7.5 'DECOR' The Decorating Materials Program

You should be able to understand most of this program, from the

previous descriptions. The first few lines simply set up the LOCAL variables. Notice the use of two string arrays, to hold information for subsequent display on the screen along with the answer. The Q$=CHR$(63) line simply makes 'Q$' hold the character for a question mark, for use when prompting for an input.

Following the 'NEW::' label, we have the part of the procedure that prompts and gets in the required room information. Note that, for the length and width, the procedure expects the measurement to be entered in feet (and a *decimal* part of a foot), but for the height, it is entered in inches. You can change this, if you wish. Note the length of the prompt strings: even with the question mark, they are kept to less than 16 characters. This is to prevent the display from scrolling.

After the measurements have been entered, the calculations are made of the perimeter, and the wall and ceiling areas. The first Menu is then offered, and this is followed by a check on which of the options has been selected. The first check (IF PE%=0) tests whether the **CLEAR/ON** key has been pressed, returning to the Organiser Menu if it has been. If this were not done, an error could arise later on in the program, since the value of the variable 'PE%' is used to select an element in a string array – and this must not be '0'.

The routine to handle 'Emulsion' follows – starting with the input of the coverage per gallon, and then, via the 'Ceiling or Walls' Menu procedure, continuing with the appropriate calculation. Should this part of the procedure be selected when it is run, program control will jump to the last 'ENDIF' after either calculation has been made, ready to obey the 'display' set of instructions.

The routine for 'Paper' follows a similar pattern. Note that it is not necessary to use an 'ELSEIF PE%=1' instruction: if PE% is not equal to zero, and not equal to 2 (the previous tests), it can only be equal to '1' since no other values are possible as a result of the MENU instruction. Hence an 'ELSE' can be used.

The necessary calculations having been made, the screen is cleared and the selected options (CEILING or WALL, PAPER or EMULSION) are displayed on the top line, by using the result of the Menu selections (CW% and PE%) to identify the required string array elements. The semi-colon at the end of this instruction line prevents scrolling of the screen, which could happen when the selections result in 'CEILING EMULSION' being displayed. The next line 'AT 1,2' is to ensure that the cursor is always positioned at the start of the second line, for the display of the answer.

The answer is preceded by a message which depends on the choice of materials. Note the calculation for Emulsion:

ANS%/8;"GALS + ";ANS%-(8*(ANS%/8));"PINTS"

Although we have calculated how many *pints* are required, it is more useful to know how many gallons are needed. This is achieved

through the 'ANS%/8' calculation. The result will be an integer – that is, the decimal part will be 'lost'. To get the decimal part, in terms of pints, we take the total number of pints, and deduct from it the number of pints there are in the number of gallons already accounted for. The number of gallons already accounted for is 'ANS%/8': hence eight times this value is the number of pints already accounted for. This value is deducted from the total number of pints needed, to give the additional number of pints. Thus, instead of giving an answer such as 17 pints or 2.125 gallons, we get a more meaningful answer of 2 gallons 1 pint.

You can adapt this program to suit your own needs: for example, you may decide that you would rather calculate the number of *litres* of emulsion required, rather than gallons/pints. This would involve a simple change to the 'CALEM%' procedure (there would be no need to convert from coverage per gallon to coverage per pint), changing the message "SQ.FT PER GAL" to "SQ.FT PER LITRE", and changing the display of the answer to 'PRINT ANS%;"LITRES" – cutting out all the conversion work to gallons and pints.

3.8

MORE KEYBOARD AND SCREEN HANDLING

OPL words covered
KEY, KEY$, PAUSE, KSTAT, CURSOR ON/OFF, VIEW

Has a key been pressed?

We have discussed two OPL words which enable information to be typed in from the keyboard. These are 'GET', and 'INPUT'. Let us look now at exactly how these two words work.

Organiser has a special area in RAM called the *input buffer*. When keys are pressed, Organiser places the information about those keys into the input buffer area. *It will do this as an* ***interrupt*** *operation.* (See Chapter 1.3, "*Excuse the interruption*"). Thus, if you press keys whilst Organiser is performing a lengthy operation – and is not 'expecting' an input from the keyboard – the information about the pressed keys will be stored in the input buffer.

When Organiser meets a word requiring an input from the keyboard – such as 'GET' or 'INPUT' – it looks *first* into the input buffer area, to see if any keys have already been pressed: if it finds information there, it uses that information for the instruction, and *removes it from the input buffer.*

If it doesn't find information in the input buffer it assumes that, so far, a key has not been pressed and it waits at the keyboard to obey the instruction.

'GET', for example, causes Organiser to look into the input buffer to see if a key has been pressed. If more than one key has been pressed before the GET instruction is been reached, it takes the information relating to the first key (GET accepts just one key-press), and associates that with the GET instruction. If the instruction had been KEYIN%=GET, it puts the pattern number or ASCII value for the pressed key into the boxes reserved for the variable 'KEYIN%'. The key information is then removed from the input buffer: it is no longer needed.

'GET$' operates in a similar way except that this time, instead of the input information being regarded as a *value*, it is regarded as a *character*. You cannot mix the variable types: 'K$=GET' would produce a **TYPE MISMATCH** error during the procedure translation process.

'INPUT' accepts key-presses until it meets an **EXE**: in this instance, it takes any values in the input buffer up to the **EXE** 'character', and places those in the variable associated with the INPUT instruction. At the same time, it displays each of the values,

as a character, on the screen.

If Organiser finds the input buffer empty, it waits at the keyboard until keys are pressed. It then places the information into the input buffer and acts accordingly as just described.

There are two OPL words that look *only* in the input buffer, to see if a key has been pressed: if the input buffer is empty, then Organiser doesn't wait for a keypress – it goes on with the next instruction. These words are 'KEY' and 'KEY$'. For both of these words, you must tell Organiser where to store any information it may find. The complete instructions therefore take the form:

K% = KEY
K$ = KEY$

KEY

When Organiser sees this instruction, it looks in the input buffer and if it sees that one or more keys have been pressed, it takes the pattern number or ASCII value of the *first* keypress, and stores it in the allocated variable.

If it finds that no key has been pressed – the input buffer is empty – it stores '0' in the allocated variable, and continues with the next instruction.

KEY$

This is similar to 'KEY' except that, this time, the *character* for the pressed key is stored in the allocated variable – which must of course be a 'string' type.

If it finds that no key has been pressed, then it stores a *null* character – in other words, nothing – which, in computer language can be represented as "". """" means 'an empty string'.

These instructions are used when you don't want to have Organiser wait at the keyboard for a keypress – but you do want to know if a key has been pressed, and if so, which one. (This will be demonstrated in a program later on in this Chapter).

They are particularly useful for 'games' type programs, where Organiser will be required to 'do something' – change the display, for example – while waiting for a key to be pressed, and for checking whether one of the *control* keys has been pressed.

These control keys – **CLEAR/ON**, **MODE**, **EXE**, **DEL** and the cursor keys – do not produce a *character*, but nevertheless are allocated a character pattern number. When one of these keys is pressed, the corresponding pattern number is placed in the input buffer before it is acted upon. The 'KEY' instruction can be used to examine the input buffer to see whether a control key has been pressed during the execution of previous program instructions, just as it can be used to see whether any other key has been pressed.

The pattern numbers associated with the control keys are

1 **CLEAR/ON** key.
2 **MODE** key
3 **UP** cursor key
4 **DOWN** cursor key
5 **LEFT** cursor key
6 **RIGHT** cursor key
7 **SHIFT** and **DEL** keys, together.
8 **DEL** key
13 **EXE** key

These pattern numbers will also be returned if the corresponding key is pressed during a GET instruction. During an INPUT instruction, the *effect* of the key is obeyed if it is a permitted operation – **DEL** for example will delete from the input buffer the information for the character immediately before the cursor position.

Hang on a moment ...

So far, when we wanted to retain a display on the screen, we used the 'GET' instruction on its own. This tells Organiser to check if a key has been pressed and, if not, to wait until a key *is* pressed. Processing then continues with the next instruction.

This means that the display stays on the screen until a key is pressed. But you may not always want to press a key to move the processing on: you may wish to leave the display on the screen for a few moments, and then have the next instruction obeyed *automatically*.

There is a word in OPL that allows you to do this: it is 'PAUSE'.

PAUSE

must always be followed by an integer value or variable. The value represents a number of *twentieths* of a second. Thus a value of '10' represents half a second, '20' represents one second, and '100' represents 5 seconds.

If the value after 'PAUSE' is greater than zero, Organiser will simply wait for the specified amount of time before performing the next instruction. Thus PAUSE 20 tells Organiser to wait for one second, then continue with the next instruction.

If the value after 'PAUSE'is negative (less than zero), then Organiser will wait for the specified amount of time *or* until a key is pressed – whichever comes first. As with the 'GET' and 'INPUT' instructions, Organiser looks in the input buffer first. If it finds a key has been pressed, it doesn't wait at all. 'PAUSE -100', for example, will cause Organiser to wait for five seconds

before continuing with the next instruction – or to continue immediately Organiser finds that a key has been pressed. If a key is pressed *before* the PAUSE started or during the execution of the PAUSE instruction, the pattern number or ASCII value for the pressed key is *left in the input buffer.*

If the value is '0', then Organiser simply checks the input buffer and, if it is empty, waits until a key is pressed – the pattern number or ASCII value for the pressed key is then left in the input buffer. The difference between PAUSE 0 and GET is that, with GET you can tell Organiser what to do with the inputkey information in the same instruction. Thus KEYIN%=GET, puts the pattern number for the pressed key into the variable 'KEYIN%'. With PAUSE 0, if you wanted to know *which* key had been pressed *while the* PAUSE *instruction is being executed*, you would have to use the 'KEY' instruction to examine the input buffer.

If you simply want to keep the display on the screen until a key is pressed, you can therefore use either 'PAUSE 0' or 'GET'. However, because the keypress information is left *in the input buffer*, a subsequent 'PAUSE', 'GET' or 'INPUT' instruction will act on that information. Here is a short procedure to demonstrate the use of 'PAUSE' with a negative value.

```
WAIT:
PRINT "I AM PAUSING..."
PAUSE -40
PRINT "OFF I GO AGAIN"
GET
GET
```

Program 3.8.1 'WAIT' Demonstration of 'PAUSE' (1)

When run, Organiser holds up processing for 2 seconds (40 *twentieths* of a second) *or* until a key is pressed, whichever comes first. If a key is pressed *during* this 2-second period, the ASCII value for the key will be placed in the input buffer, and Organiser will continue with the next instruction 'PRINT "OFF I GO AGAIN"'. It will then arrive at the first 'GET' instruction. It looks in the input buffer – and finds information already there. So it moves onto the next instruction – another 'GET'. This time, the input buffer is empty, and so Organiser once again waits for a key to be pressed, before it exits the procedure.

If a key is *not* pressed during the 2-second delay period, then after the 2 second delay, Organiser will continue automatically. This time, when it reaches the first 'GET' there will be *no* information in the input buffer, and so it waits for a keypress before going onto the next instruction – another 'GET'. In these circumstances, *two*

keypresses will be necessary to complete the procedure.

If you have entered this procedure, you can EDIT it so that it reads as follows, to test the operation another way. (Alternatively, ERASE the previous procedure and re-enter the new one). This procedure also demonstrates the use of the 'KEY' instruction, to examine the input buffer.

```
WAIT:
LOCAL K$(1)
PRINT "I AM PAUSING"
PAUSE -100
K$=KEY$
CLS
IF K$=""
  PRINT "OFF I GO AGAIN"
ELSE PRINT "YOU PRESSED ";K$
PRINT "WHILE I PAUSED"
ENDIF
GET
```

Program 3.8.2 'WAIT' Demonstration of PAUSE (2)

This time, if a key is pressed during the 5 second delay period resulting from 'PAUSE -100', it is retained in the input buffer, and the next instruction 'K$=KEY$' is *immediately* obeyed. This examines the input buffer, sees a key has been pressed, and places the *character* in K$. The screen is then cleared ('CLS'), and a test is made to see whether anything has been stored in K$ ('IF K$=""'). There *will* be something in K$ – so it will not be 'null' and the comparison is not true. Organiser then moves on to the 'ELSE' instruction to PRINT the message "YOU PRESSED " followed by the character for the key you pressed, and the rest of the message. The input buffer will have been cleared by the 'K$=KEY$' instruction, and so when Organiser reaches 'GET', it waits at the keyboard for you to press a key – and the display remains on the screen.

If a key is *not* pressed during the 5 second delay period, at the end of it Organiser continues automatically with the next instruction. This time, 'K$=KEY$' results in a null string ("") being stored in K$, so that the comparison in the next instruction will be *true*. Consequently Organiser will obey the next instruction to PRINT "OFF I GO AGAIN", and will then (via ENDIF) obey the GET instruction as before.

Setting up the keyboard

When the 'INPUT' or 'GET' instruction is used, the keyboard is automatically set for the associated variable type. 'INPUT NUMBER' or 'INPUT NUMBER%', for example, sets the keyboard for *numeric* inputs, whilst 'INPUT NUMBER$' sets the keyboard for

character inputs. During *character* inputs – to 'INPUT NUMBER$' for example – you may want to ensure that *numbers* are input. You may be entering a telephone number, for example. Organiser's keyboard will be set for *character* inputs, and so you would have to remember to press the **SHIFT** key whilst making your number entry.

There is an alternative. You can set the keyboard to give any of the possible inputs by using the 'KSTAT' (short for 'Keyboard STATus') instruction. This must be followed by a number from 1 to 4. The keyboard is set accordingly:

KSTAT 1 Sets the keyboard for *capital* letter inputs.
Pressing **SHIFT** and a key will produce the *numeric* character printed above the key.

KSTAT 2 Sets the keyboard for *lower case* letters.
Pressing **SHIFT** and a key will produce the numeric character printed above the key.

KSTAT 3 Sets the keyboard for numeric inputs.
Pressing **SHIFT** and a key will produce the *capital* letter displayed on the key.

KSTAT 4 Sets the keyboard for numeric inputs.
Pressing **SHIFT** and a key will produce the *lower case* letter displayed on the key.

Thus, you can ensure that the keyboard is set to the type of input you want in your program.

The keyboard will remain in the 'KSTAT' state you set it, even if Organiser is switched off. The operation is the same as if the **SHIFT** and **CAP** or **SHIFT** and **NUM** keys had been pressed. You can restore the state of the keyboard by using these keys, in the usual way.

When an 'INPUT NUMBER%' or 'INPUT NUMBER' type of instruction is used, Organiser expects an actual number to be entered at the keyboard. The keyboard is consequently automatically set for numeric inputs – you don't have to use the 'KSTAT' command. However, it is possible to make a *non* numeric input – by pressing the A key, for example. This would result in the < symbol (the numeric character above the 'A' key) being displayed. When **EXE** is pressed, to complete the entry, Organiser will be puzzled: it is expecting a *number*, and it has found a *symbol*. Consequently, it will display a question mark, on the next line, and wait again for you to enter a *number*.

When 'GET' is used, Organiser looks at the pattern number or ASCII value for the pressed key. This time, Organiser leaves the keyboard in its current state: if you want to GET the 'pattern number' for a numeric key, a capital letter or a lower case letter –

and don't want to have to use the **SHIFT**, **CAP** or **NUM** keys in order to make the input, then the KSTAT instruction is the answer. The following procedure will allow you to experiment with this.

```
KTEST:
LOCAL PN%,K%
MORE::
CLS
PRINT "KSTAT NO (1-4)";CHR$(63)
INPUT K%
IF (K%<1) OR (K%>4)
  GOTO MORE::
ENDIF
KSTAT K%
CLS
PRINT "PRESS A KEY"
PN%=GET
CLS
PRINT "ASCII NO OF ";CHR$(PN%)
PRINT "IS ";PN%
GET
PN%=MENU("MORE,END")
IF PN%=1
  GOTO MORE::
ENDIF
```

Program 3.8.3 'KTEST' Keyboard input tester

This procedure should be fairly easy for you to understand. After entering the required value for the 'KSTAT' instruction (via K%), it is tested to ensure it is within the permissible range of 1 to 4. If it isn't, processing jumps back to 'MORE::' – to get the input again. Once a key has been pressed, the screen is cleared and the message "ASCII NO OF " followed by the character for the keypress is displayed, and the actual ASCII value or pattern number. Pressing a key will then give you the option of further experimentation, or ending. Note that it is necessary to test only if PN% = 1 for the end Menu: we want any other value to exit the procedure – which Organiser will do if it reaches the last 'ENDIF' instruction. You will notice too that it is not necessary to use a 'RETURN' instruction if it will be the *last* instruction in the procedure – and a value is not being returned.

Using the procedure, you can test out the ASCII values for the control keys: there will be no (sensible) character patterns for these keys, of course.

Where's the cursor gone?

You may or may not have noticed that, when an 'INPUT' instr-

uction is used, a cursor is displayed on the screen – either a blob or an underline, depending on how the keyboard has been set. When a 'GET' instruction is used, however, the cursor is not normally displayed.

It may be that you would *like* the cursor to be displayed, perhaps to indicate that Organiser is waiting for an input. There is an OPL instruction that allows you to display the cursor on the screen. It is (surprise, surprise) 'CURSOR ON', and is a complete instruction in itself.

You could add the 'CURSON ON' instruction in Program 3.8.3 – immediately after the declaration of variables, for example – so that the cursor will be displayed when the 'PN%=GET' instruction is reached. Once 'CURSOR ON' has been used, the cursor will be displayed whenever a 'PAUSE' instruction is being executed, as well as a 'GET' instruction.

To remove the cursor display from the screen during 'GET' and 'PAUSE' instructions, the instruction is 'CURSOR OFF'. Note that, even if 'CURSOR OFF' is used, the cursor will still be displayed for 'INPUT' instructions. That's so you can see what you're doing when entering a series of characters.

Viewing a string

The 'MENU' instruction allows you to set up your own Menu, and to devise a course of action according to the selected option: a pressed key returns the *position* of the option within the Menu string. Commas used to separate the options in the string are not displayed, and the option words are displayed if necessary on separate lines.

There is another OPL instruction – VIEW – which allows you to display a *string* on the screen. The full instruction looks like this:

KEY% = VIEW(line%,string$)

With 'VIEW', the string is displayed character by character as *written* – and if it is longer than sixteen characters, then the display will scroll to the left. This scrolling can be controlled by the cursor keys. The string can, of course, be an actual string, enclosed within quotation marks, or a string variable.

The line% value – an actual number or an integer variable – determines the line of the screen on which the string will be displayed. It must be either a '1' for the top line, or a '2' for the bottom line.

When a key *other* than a cursor key is pressed, the ASCII value of that key is returned. This could be used to determine a course of action, just as with the MENU instruction.

However, the VIEW instruction is intended (as the name suggests) more for *viewing* string variables – a file record, perhaps.

Also, like the GET instruction, it can be used *without* allocating the key-press information to an integer variable. Thus

VIEW(1,ADDRESS$)

is a valid instruction. In this instance the display will remain on the screen until a key is pressed – scrolling, if it is longer than sixteen characters.

The value of this instruction will become far more apparent when we deal with the handling of *files* – an extremely powerful feature of Organiser.

If a second 'VIEW' instruction is used, the second one can use '0' for the line number: however, if a different string is used for the second instruction, strange displays can result – a mixture of the two strings with an additional unusual character, perhaps. It is probably better to define the first or second line for subsequent 'VIEW' instructions, rather than use the '0' facility.

3.9

LOOPING ROUND UNTIL THE JOB IS DONE

OPL words covered
DO/UNTIL, WHILE/ENDWH, CONTINUE, BREAK, REM

Keep going until...

There are many occasions when you will want to repeat a series of instructions until certain conditions are met. This *could* be achieved by having labels at the beginning and possibly at the end of the instructions that need to be repeated, and using an 'IF' test within the instructions. Depending on the result of the test, you would either jump back to the label at the beginning of the instructions, or to the label at the end of the instructions.

For example, you may wish to write a program which calculates the annual interest on an invested sum of money, and see how the sum grows each year for five years. The framework for such a program could look like this:

```
INTEREST:
  YEAR=1
  REPEAT::
  Calculate interest on capital
  Add interest to capital (=new capital)
  Display answer (and hold it)
  Add 1 to year
  IF YEAR=6
  GOTO DONE::
  ELSE GOTO REPEAT::
  ENDIF
  DONE::
```

This method will work, but Organiser provides two better methods requiring less space and less programming effort. There are two sets of OPL words which enable a set of instructions to be performed in a 'loop' while or until specified conditions are met.

One set – 'WHILE/ENDWH' (short for 'ENDWHile') – tests the condition for obeying the set of instructions *at the beginning*, while the other set – 'DO/UNTIL' – tests the condition for obeying the instructions *at the end*.

Thus, with 'WHILE/ENDWH', if the test condition is *not* true, the instructions are not obeyed. With 'DO/UNTIL', however, since the test is made at the end, the instructions will always be obeyed at least *once*.

The difference between these OPL words is quite small, and in many instances a program can be written using either set.

WHILE a condition is true...

The WHILE instruction must be followed by a comparison statement (just like the IF instruction). It says to Organiser, "WHILE the comparison is *true*, perform all the instructions that follow, until you get to the ENDWH instruction. Then come back to the WHILE instruction – and start again".

Clearly, the in-between set of instructions must change the values in the comparison test in some way – otherwise they will be repeated indefinitely. Organiser will go into a continuous loop.

To demonstrate the use of this construction, let us create a short procedure to display a multiplication table. We will allow the user to enter any number (even decimal values), then Organiser will display that number multiplied by 2, 3, 4 – and so on – up to 12. If we call the input number 'N' (*a floating point value*), and the multiplier 'M%' (*an integer number*), then we want the procedure to run for as long as M% is less than 13.

```
MTABLE1:
LOCAL N,M%
M%=2
PRINT "ENTER NUMBER"
INPUT N
WHILE M%<13
  CLS
  PRINT N;"x";M%;"=";N*M%
  M%=M%+1
  PAUSE 30
ENDWH
PRINT "DONE"
GET
```

Program 3.8.1 'MTABLE' Multiplication Table using WHILE/ENDWH.

Note that, like the IF instructions, *indents* are used to help identify the instructions that must be obeyed within the WHILE/ENDWH loop: you do *not* need to enter the indent spaces when typing in procedures.

When this procedure is run, M% is initially set to '2', and so the WHILE comparison (M%<13) is *true*. Within the WHILE/ENDWH loop there is an instruction 'M%=M%+1'. When ENDWH is reached, Organiser immediately branches back to the WHILE instruction, and again tests the comparison. This time, M% is equal to '3' – which is still less than 13. And so the instructions are obeyed again. While M% is less than 13, the process continues. When M% becomes equal to 13, the comparison 'M%<13' will be *untrue*, and so

Organiser jumps immediately to the instruction following ENDWH.

If the line 'M%=2' were to be re-written as 'M%=13', then the instructions between WHILE and ENDWH *would not be obeyed even once.*

Now let us look at the other pair of OPL words.

DO until a condition is true ...

With this pair of OPL words, 'DO' is a complete instruction (although it *can* be followed by another instruction, on the same line, without need for the 'space and colon'). It tells Organiser to obey the instructions that follow, until it comes to an UNTIL instruction.

The UNTIL instruction must be followed by a comparison, just like IF and WHILE. This time, if the comparison is *not true*, Organiser goes back to DO and repeats the instructions down to UNTIL again. Clearly, as with WHILE/ENDWH, the conditions for the comparison must be changed during the set of instructions to be obeyed – otherwise Organiser will keep going round and round the instructions, ad infinitum.

Because the comparison test is not made until *after* the instructions have been obeyed, they will always be obeyed at least *once*. Let us look at the multiplication table again, this time using the DO/UNTIL words to control the loop.

```
MTABLE2:
LOCAL N,M%
M%=2
PRINT "ENTER NUMBER"
INPUT N
DO CLS
  PRINT N;"x";M%;"=";N*M%
  M%=M%+1
  PAUSE 30
UNTIL M%>12
PRINT "DONE"
GET
```

Program 3.9.2 'MTABLE2' Multiplication table using DO/UNTIL.

Apart from the DO/UNTIL instructions, you will notice that the comparison is different this time: instead of performing the instructions *while* M%<13, we perform them *until* M%>12.

Why didn't we say 'UNTIL M%=13'? That would work, but it *could* lead to problems in larger programs. If, for some reason, the instructions caused the value of M% to become *greater than* 13, then the condition 'M%=13' would *never* be met – and Organiser would go looping the loop indefinitely. By making the comparison 'M%>12', we have allowed for M% taking on *any* higher value than

is required. In this procedure, that eventuality is extremely unlikely (if not impossible!). Nevertheless, it is good programming practice to protect against potential running errors.

This procedure will produce the same display as the 'MTABLE1' procedure. However, this time, if the line M%=2 is changed to M%=13, then the instructions *will* be obeyed once – because Organiser 'sees' them all *before* it reaches the test comparison at the UNTIL instruction – and the display will show the entered number multiplied by 13 before the message 'DONE' appears.

Nesting loops

Just as the IF instructions can be 'nested' – that is one set of IF instructions can be included within another set – so WHILE/ENDWH and DO/UNTIL instructions can be nested.

You can have as many as eight loops (and IF instruction sets) nested within each other – far more than you will normally require even for the most complex procedure. Each 'loop' must be properly terminated with an ENDWH or an UNTIL instruction, otherwise you will get a **STRUTURE ERR** message during the TRANslation process. With such an error, when you press **SPACE** to re-enter the procedure for editing, Organiser will generally be marking a line near the end of the procedure, and not necessarily at the place where you want the terminator instruction to appear.

Breaking out of a loop

We have seen that the instructions between both WHILE/ENDWH and DO/UNTIL are repeated *while* or *until* the test comparison is true. You may, however, want Organiser to break out of the loop if *another* condition is met – perhaps if a particular key is pressed, or if a specific value has been found. Or you may want Organiser to miss out the loop instructions for as long as the additional condition is met.

There are two OPL words to help you: 'BREAK', and 'CONTINUE'. Usually these instructions – which are complete in themselves – will follow an IF instruction which tests for the required condition.

BREAK

This causes Organiser to leave the loop altogether, and to continue from the instruction *following* the loop terminator (ENDWH or UNTIL).

CONTINUE

This causes Organiser to jump to the *test comparison* for the loop, missing out all the instructions between CONTINUE and the loop terminator. If the *test comparison* still requires the loop to be repeated, it will be repeated. *It is important that any*

instructions changing the result of the loop comparison test are ***not*** *amongst those that are skipped as a result of* CONTINUE – otherwise the condition for performing the loop will never change, and it will be repeated endlessly.

Let us take a simple example by way of demonstration. The following procedure will display all of the characters available within your Organiser. These have pattern numbers from 32 to 127, and from 160 to 255. We will use the WHILE/ENDWH type of loop, and we will allow for you to end the program whilst it is running, by pressing the **DEL** key (Character 'pattern' number 8). Should you wish to stop temporarily at the display for any specific character, press the **CLEAR/ON** key. To restart the program again, press any key except **Q** – which quits the program altogether. This is a built-in feature of Organiser, and doesn't need any programming.

Note that the pattern number is updated *before* the test of whether the number is outside the required range.

```
CHARDIS:
LOCAL C%,K%
C%=31
WHILE C%<255
  C%=C%+1
  CLS
  K%=KEY
  IF (C%>127) AND (C%<160)
    CONTINUE
  ELSEIF K%=8
    BREAK
  ENDIF
  PRINT "PATTERN NO ";C%;"=";CHR$(C%)
  PAUSE 10<-
ENDWH
IF C%>254
  PRINT "DONE"
ELSE PRINT "YOU STOPPED ME"
ENDIF
GET
```

Program 3.9.3 'CHARDIS' Display of character patterns

When you run this procedure, you will notice that the screen blanks out for a fraction of a second when the Pattern Number reaches 128, and returns with the display for Pattern number 160. Should you wish to have the patterns on display for a little longer, change the value following PAUSE.
It is now time to introduce another OPL word that actually does nothing!

Making a REMark

It can be very useful when writing and developing procedures and programs to incorporate a 'note' to yourself – a reminder of what you have done or the purpose of a particular set of instructions, perhaps.

The OPL word for such a note is 'REM' – short for 'REMark'.

You use REM as if it were an instruction, following it by any message that you may want. When Organiser sees REM during the TRANslation process, it skips on to the next genuine instruction, ignoring the REM and anything that is incorporated within the same instruction line. Thus, although the REM line occupies memory space in the OPL procedure, the translated procedure takes up no more space.

The use of REM helps to make programs easier to follow when you are 'debugging' or changing them, and enables 'markers' to be placed in the program to indicate where you may wish to add instructions at a later stage.

In the game program that follows, for example, REM is used extensively to explain the function of each instruction line.

A Target game program

The game about to be detailed encompasses most of the techniques discussed so far. Indeed, it has been written to do just that. For example, it uses the BREAK instruction to allow you to stop playing at anytime – and get the score to date – by pressing the **EXE** key.

It also introduces some new techniques – in particular, 'computer animation'. Whilst you may not be into 'game playing', it would be a worthwhile exercise to study the program and to enter it, even if it is erased at a later date.

In the game, arrows will be made to travel across the screen, from left to right, at increasing speeds. The object of the game is to stop the arrow with a 'bat', controlled by the **UP** and **DOWN** cursor keys. Two levels of play are provided. It is a very simple game, but nevertheless will serve our purposes well.

The flowchart for the game is shown in Fig. 3.9.1: in this chart, you will notice that there are 'decision points'. All of these *could* be written using the IF instruction: however, the program demonstrates how DO/UNTIL and WHILE/ENDWH instructions can be brought into use to cut down the number of programming lines required.

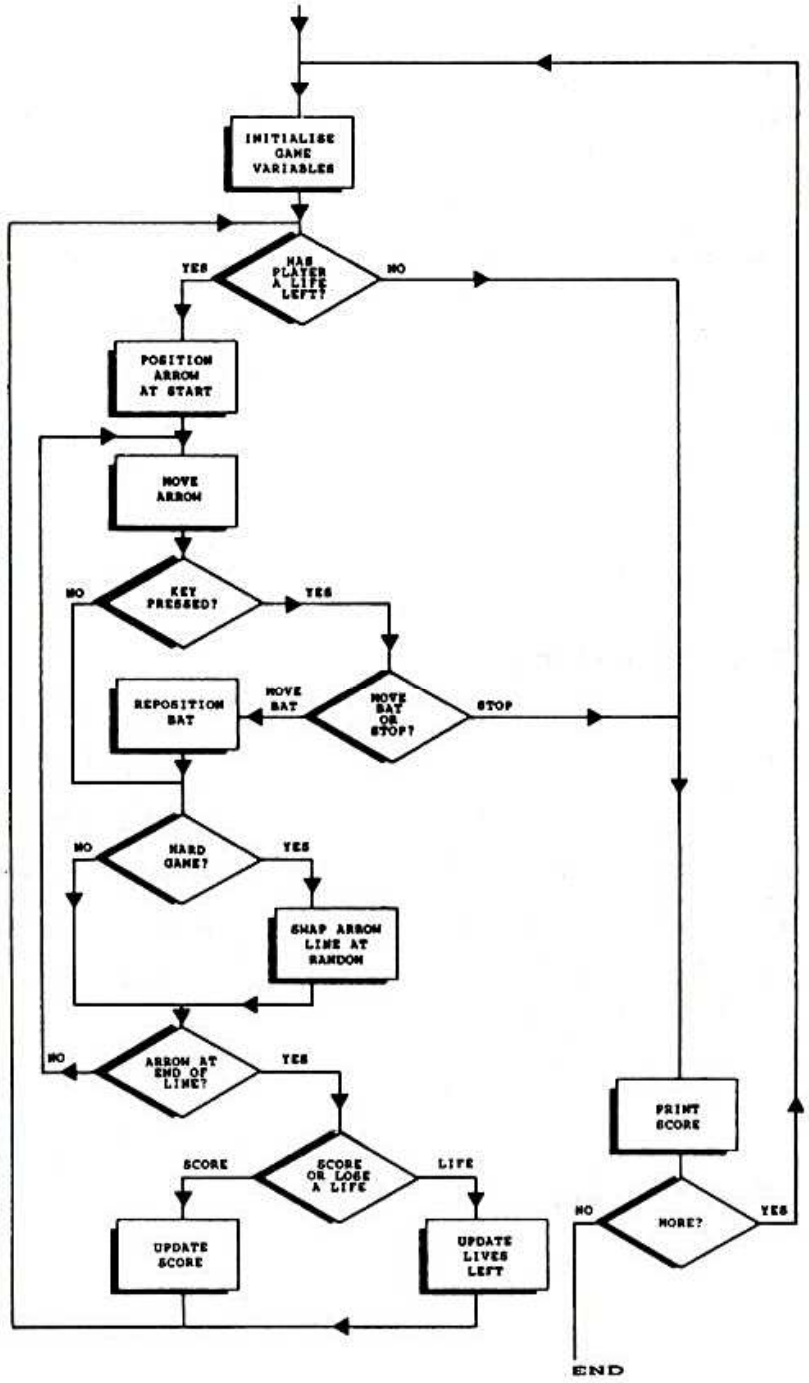


Fig. 3.9.1 Flowchart for the 'Target' game.

The game is written in one main procedure and two short procedures. We will deal with the short procedures first.

During 'game play', we will require the arrows to be shot across the screen at random intervals – and on either the top or bottom line, also at random. The first procedure therefore returns a random number up to a specified limit – the limit being passed into the procedure as a parameter.

```
RND%:(N%)
RETURN 1+RND*N%
```

Program 3.9.4a 'RND%' Random numbers for Target game

(Remember you press **EXE** after entering the procedure name 'RND%', and then enter the parameter information – don't try to enter the parameter information as part of the title). The **%** symbol at the end of the procedure name ensures that an *integer* will be returned from the procedure – so that all the decimal parts of the number generated by RND will be ignored automatically, without having to use the INT instruction.

Now for the second procedure. We want the arrows to start off by traversing the screen at a fairly modest speed, and to get faster as the game progresses. The shortest time delay that can be obtained with the PAUSE instruction is one twentieth of a second: if we moved the arrow at one-twentieth of a second for each character position, it would take nearly a second to traverse the screen – and that is much too slow.

The second short procedure provides a delay that depends on the value of an input parameter to the procedure. It uses a simple DO/UNTIL loop – which takes time to execute, but nevertheless is considerably faster than can be achieved by using PAUSE. We also want the arrow to be 'fired' at random intervals (as determined by the 'RND%' procedure): we could use the PAUSE instruction for this, but since we need a 'DELAY' procedure anyway, we shall make full use of it.

```
DELAY:(P%)
LOCAL C%
C%=0
DO
  C%=C%+1
UNTIL C%>P%
```

Program 3.9.4b 'DELAY' periods for Target game.

The line C%=0 is not actually needed in this procedure: when a variable is declared, it is automatically set to zero. The line is included for completeness, to help you understand how the procedure operates. The procedure simply keeps adding one to the variable C%, until it is one greater than the input parameter P%.

Each 'loop' takes only thousandths of a second to execute, but the overall delay – depending on the value of P% – will be sufficient for our needs.

Now for the main procedure. This uses eight LOCAL variables: to make it easier for you to enter, these are declared as single letters. They are used to store information as follows:

X% The position of the arrow across the screen.
Y% The line of the screen on which the arrow will appear.
L% The number of 'lives' left – you are allowed three, unless you assign a higher number to L%!
B% The line of the screen on which the bat is to be displayed.
D% The delay period for each movement of the arrow.
M% The player's move of the bat.
S% The score, displayed after the game is over.
H% The player's choice of a hard or an easy game.

The program includes 'REMarks': you do *not* need to enter these – in fact to do so will consume a lot of memory space unnecessarily. They make no difference to the running of the program: they are included simply to help you understand how the program operates. Where a REM is included on the same line as another instruction, 'spaces' and a colon separate REM from that instruction: these should not be entered either.

Also, as before, you do *not* have to enter the spaces for leading indents: they are there to help you see the program structure.

Finally, you will see a new OPL word in this program – 'BEEP': this is to add sound effects to the game. BEEP is covered in full detail in a separate Chapter.

```
TARGET:
LOCAL X%,Y%,L%,B%,D%,M%,S%,H%
START::
H%=MENU("HARD,EASY")
REM  Set up the game variables
B%=1                        :REM Bat on the top line
D%=50                       :REM Initial delay period
L%=3                        :REM Start with three lives
S%=0                        :REM Start with a zero score
AT 16,B%
PRINT CHR$(124)             :REM Display bat at top right
WHILE L%>0                  :REM While player has a life
  DELAY:(RND%:(500))        :REM Random delay for arrow
  X%=2                      :REM First position on screen
  Y%=RND%:(2)               :REM Random line number
  DO                        :REM Move arrow across screen-
    AT X%-1,Y%              :REM Position cursor
    PRINT " ";CHR$(126)     :REM Blank old/print new arrow
    BEEP 2,X%*100           :REM Sound effects
    M%=KEY                  :REM Key-pressed meantime?
```



```
    IF (M%=4) OR (M%=3)     :REM Down/up cursor key?
      AT 16,B%              :REM Yes-clear old bat
      PRINT " ";
      B%=M%-2               :REM New line for bat
      AT 16,B%
      PRINT CHR$(124)       :REM Print bat again
    ELSEIF M%=13            :REM EXE pressed to exit?
      L%=0                  :REM Yes - make lives zero
      BREAK                 :REM and jump out of DO loop
    ENDIF
    DELAY:(D%)              :REM Slow up arrow
    X%=X%+1                 :REM New position for arrow
    IF (X%=8) AND (H%=1)    :REM Midway and Hard game?
      AT X%-1,Y%
      PRINT " "             :REM Clear old arrow display
      Y%=RND%:(2)           :REM Get new line number
      AT X%,Y%              :REM Position cursor
      PRINT CHR$(126)       :REM and print arrow again
    ENDIF
  UNTIL X%>16               :REM until arrow at end
  AT 16,Y%
  PRINT " "                 :REM Clear arrow display
  IF Y%<>B%                 :REM Bat/arrow on same line?
    L%=L%-1                 :REM No-lose a life
  ELSE S%=S%+1              :REM Yes-increase score
    D%=D%-10-10*(D%<10)     :REM and shorten delay
    AT 16,B%                :REM Bat will be cleared too...
    PRINT CHR$(124)         :REM ...so re-display it
  ENDIF
ENDWH                       :REM Three lives now lost
CLS
PRINT "SCORE=";S%           :REM Display score
PAUSE 40                    :REM for two seconds
S%=MENU("MORE,END")
IF S%=1                     :REM More wanted
  GOTO START::
ENDIF                       :REM otherwise, all done
```

Program 3.9.4c 'TARGET' main game procedure

The 'REMarks' should make the operation of this program reasonably clear. The only line that may need further explanation is

```
D%=D%-10-10*(D%<10)
```

The first part of this instruction – D%=D%-10 – deducts 10 from the delay period, so that the arrows get progressively faster. However, we have to watch out for the value of D% becoming negative, which could happen after a few arrows have been displayed. The

second part of the instruction ensures D% is never negative. If D% is less than 10, then (D%<10) resolves to '–1' (see Chapter 3.6) which,
multiplied by '–10' is equal to +10. Hence, the value of D% is restored.

As well as testing for which cursor key is pressed, the program also checks to see if **EXE** has been pressed – indicating that the player wishes to stop. If **EXE** has been pressed, we wish to jump out of both the DO/UNTIL and the WHILE/ENDWH loops. BREAK enables us to jump out of the current DO/UNTIL loop. However, if 'lives' are left, the program will still run within the WHILE/ENDWH loop. Consequently, to ensure that this does not happen, L%, the variable containing the number of lives left, is assigned the value zero before the BREAK instruction. This ensures that the program exits *both* loops, and continues with the score display.

You will notice that, in the 'Hard' version of this game, the arrow *may* switch lines halfway across, just to make life more difficult.

Can you see the process for animating the arrow across the screen? We start by actually printing the arrow in the first position. Then, for each subsequent move, a *space* is printed at the arrow location – visually removing it from the display, and a new arrow is displayed at the screen location to the right. The visual effect is that the arrow moves across the screen from left to right. When it reaches the end, it has to be removed from the display – by overwriting it with a 'space' character.

At the end, if the arrow is on the same line as the bat, the arrow display will overwrite the bat display, and then the arrow display will be removed: hence the bat is re-displayed if the arrow and bat were on the same line.

This technique is virtually the same as that used in all the video games to provide animated movement.

3.10

CONVERTING VARIABLES

OPL words covered
ASC, CHR$, FIX$, FLT, GEN$, HEX$
INT, INTF, NUM$, SCI$, VAL

A change is needed

There are numerous occasions when information stored in one type of variable needs to be converted to another type of variable. A typical example is when a number is stored as a string: if you wish to make a calculation using the number, it must first be transferred to a numeric type of variable.

You will recall that Organiser has three types of variable:

a) *Floating Point* variables, which occupy eight memory boxes, can have a decimal value up to 12 significant figures, and can be of virtually any magnitude. Values using 'scientific notation' (e.g. '1.2345E+03', which is the same as 1234.5) are stored as floating point numbers.
b) *Integer* variables, which occupy two memory boxes, and can have values between –32768 and +32767.
c) *String* variables, which can occupy up to as many memory boxes as you allocate when writing a program, with a top limit of 255.

Organiser can perform mathematical operations *only* on numeric variables, and string operations *only* on string variables. Mathematical operations using only integer variables will produce integer variables: any decimal part of the answer will be lost. If you want to use a numeric value as part of a *string*, the way that the value is stored must be converted first. Similarly, if you want to make a calculation using a number stored as a string, it must be converted to a form that Organiser can use.

In the descriptions that follow, unless stated otherwise, parameters specified within brackets can be variables or actual values. Thus a parameter denoted as 'string$' can be a string variable or an actual string, such as "123456.7". Numeric parameters can also be *expressions* which will resolve to a value. Thus, for a numeric parameter, you could write '4' (an actual value), or 2*6 (an 'expression').

In the instruction examples, the converted information is assigned to a variable. However, it is also possible to use the PRINT instruction to display the answer on the screen, without storing it. Thus

```
C$=CHR$(65)
```

would store the letter 'A' in the string variable 'C$', whilst

```
PRINT CHR$(65)
```

would display the result of the conversion – the letter 'A' – on the screen. If you wanted to store the information as well as display it, you would assign it to a variable, and PRINT the variable. Thus

```
C$=CHR$(65) :PRINT C$
```

Converting a string to a number

Two requirements are catered for: converting *one* character to its ASCII equivalent, and converting a series of numeric characters to the equivalent floating point value. The instructions are as follows.

ASC(string$)
This returns the integer ASCII value (pattern number) of the *first* character in the string. A typical instruction would be

```
VAR%=ASC("ABC")
```

This would store in the variable 'VAR%' the value 65 – which is the ASCII value of the first character in the string, 'A'. Note that although ASCII values are always integers, the value can be assigned to a floating point variable. Thus VAR=ASC("ABC") is a valid instruction: this time, however, 65 will be stored as a floating point value.

Organiser permits an alternative form of instruction, to handle just one character. This is

```
VAR%=%A
```

This is exactly the same as the instruction VAR%=ASC("A"). The character to be converted is preceded by a **%** symbol: quotation marks are *not* required. The limitation on this instruction is, of course, that only characters accessible direct from the keyboard can be converted.

VAL(string$)
This is used to convert a number contained in a string to a floating point variable. A typical instruction would be

```
VAR=VAL("123.4")
```

This instruction assigns the value '123.4' to the floating point variable 'VAR'. If the string value is assigned to an *integer* variable, then any decimal part of the string will be ignored. Thus, if 'VAR%' were to be used in the above example, 'VAR%' would be assigned the value '123'.

The string must contain only number characters, otherwise a **STR TO NUM ERR** will occur. The string can, however, be written in scientific notation.

Converting numbers to strings

As with string to number conversions, two requirements are catered for: the conversion of one value to the character *pattern* for that value, and the conversion of a *number* into a string. Organiser allows you to convert numbers into strings in a number of ways, enabling you to determine how many decimal places will be contained in the string, to *position* the number (i.e. justify it left or right) within the string, and so on.

CHR$(ascii%)
This gives the *character pattern* for the integer value contained in the brackets. It converts a value stored in *two* memory boxes – the integer value – into a one-character string, which is stored in *one* memory box. Thus

```
C$=CHR$(65)
```

would store the *character* 'A' in string variable 'C$'.

FIX$(value,dec-places%,length%)
This instruction converts a floating point or integer *number* into a string - the *number* being the first parameter within the brackets. The number of *decimal places* is determined by the second parameter contained within the brackets, and the *length* of the string (*including* the decimal point) is determined by the third parameter. If the *length* specified is insufficient for the string to cater for the required number of decimal places plus the decimal point and the *integer* part of the value, Organiser will return a string of asterisks.

If the length% parameter has a *negative* value, the *number* will be justified at the *right* of the string.

The following examples demonstrate the use of FIX$ (in each instance, F$ must have been declared to hold at least as many characters as required by the length% parameter, of course).

```
F$=FIX$(123.456,1,5)        (F$ holds "123.5")
F$=FIX$(123.9,0,5)          (F$ holds "124")
F$=FIX$(123,2,8)            (F$ holds "123.00")
F$=FIX$(123,2,-8)           (F$ holds "  123.00")
F$=FIX$(123.9,3,4)          (F$ holds "****")
```

Notice that in the first two examples, Organiser rounds up the value before storing it into a string. In the third example, the

request was for two decimal places – and since the value to be converted had none, *two trailing zeroes* are added to the string. This feature can be useful for converting values representing money to a string for display.

The fourth example shows what happens when a *negative* length is specified: the numeric value is 'justified' to the right of a string that is eight characters in length.

The last example demonstrates what happens when an insufficent length is specified for the conversion. The number to be converted has three *integer* digits. In addition to these, three decimal places have been called for. String space is also required for the decimal point: consequently, a string length of at least seven characters is required. Four only are specified – which is not enough. Organiser shows that the number cannot be represented as a string according to the requirement, by storing asterisks.

GEN$(value,length%)
This is similar to FIX$, with one main difference. With GEN$ the number of decimal places is not specified: all decimal places in the value will be represented in the string, and trailing zeroes will not be added to 'pad out' the string, as with FIX$. Organiser tries first to
represent the value as an integer, then as a floating point, then in scientific notation.
The length% parameter can be specified as a *negative* value: this tells Organiser to 'justify' the converted value to the right. Thus if the converted value has less digits than specified by the length% parameter, the leftmost end or start of the string will be filled with spaces. Typical examples are

```
G$=GEN$(45.6,6)          (G$ holds "45.6")
G$=GEN$(45.6,-8)         (G$ holds "    45.6")
```

In the second example, there would be four spaces in the string before the first digit.

NUM$(value,length%)
This is similar to GEN$, the difference being that the value is turned in to an *integer* before it is converted to a string. Thus

```
N$=NUM$(123.456,5)
```

would result in 'N$' holding the string "123". If a *negative* length% is specified, the string will be right justified. If the integer value has more digits than specified by the length% parameter, the resulting string will contain just asterisks. Thus:

```
N$=NUM$(123.456,5)       (N$ holds "123")
N$=NUM$(1.2,-5)          (N$ holds "   1")
N$=NUM$(12345.6,4)       (N$ holds "****")
```


SCI$(value,dec-place%,length%)
This is the same as FIX$, except that the value is changed to *scientific notation* before being converted into a string. The length% parameter must be *at least six* greater than the dec-place% parameter to allow for the integer part, the decimal point, and the 'E+00' part. Otherwise a string of asterisks will be stored. If a negative length% is used, the resulting representation is right justified within the string. Typical examples are

```
S$=SCI$(123456,2,8)      (S$ holds "1.23E+05")
S$=SCI$(12.3,12)         (S$ holds "1.200E+01")
S$=SCI$(12.3,-12)        (S$ holds "    1.200E+01")
S$=SCI$(1,2,3)           (S$ holds "***")
```

HEX$(decimal%)
This is one for experienced programmers. It returns a string containing the *hexadecimal* value equivalent to the decimal value. Thus

```
H$=HEX$(32)
```

would result in H$ holding the string "20" – the hexadecimal equivalent of decimal value 32. Positive decimal values must lie within the range 0 to 32767 to give hexadecimal values from 0 to '7FFF'. For decimal values from 32768 to 65535 (the upper limit), deduct 65536 from the required value. Thus, to display the hexadecimal equivalent of 65520, you would enter

```
PRINT HEX$(65520-65536)
```

The hexadecimal number displayed will be 'FFF0' – equivalent to the decimal value '65520'.

In other words, *negative* values from –32768 to –1 will return hexadecimal values from '8001' to 'FFFF'. Thus it is possible to write a short procedure to convert any decimal value from 0 to 65535 to hexadecimal (note the use of a floating point variable to cater for the higher numbers).

```
HEX:
LOCAL I
PRINT "Decimal=";
INPUT I
PRINT "HEX$=";
IF I<32768
  PRINT HEX$(I)
ELSEIF I<65536
  PRINT HEX$(I-65536)
```

```
ELSE PRINT "Over FFFF"
ENDIF
GET
```

Program 3.10.1 'HEX' Converting decimal to hexadecimal.

Converting one numeric type to another

As well as being able to convert numbers to strings and strings to numbers, you can also convert one type of numeric variable to another.

FLT(integer%)

This converts an integer value into a floating point value. It would be used mainly with integer *variables*, since actual integer values can be made into floating point values by merely adding a decimal point. Thus '4' is regarded as an integer value – stored in *two* memory boxes when programming – and '4.' is regarded as a floating point value, stored in *eight* memory boxes. FLT allows two integer values to be added together when the result would be outside the normal integer range. A typical instruction would be

```
VAR=FLT(intgr1%+intgr2%)
```

If you have a procedure that is expecting a floating point parameter, but wish to pass an *integer* value to that procedure, you would use the FLT instruction to convert the integer to floating point.

INT(value) and INTF(value)

These instructions convert a floating point value to an integer value. For floating point values in the range –32768 to +32767, you would use the INT instruction. A typical example would be:

```
I%=10*INT(value/7)
```

Floating point values outside of this range cannot be stored in two memory boxes – so if just the integer part of the number is required, the INTF instruction must be used:

```
I=INTF(value)
```

The INTF instruction can be used for *any* value – including those within the range –32768 to 32767. However, the resulting number is stored as a *floating point* variable – with the decimal part of the value removed. If you had a procedure that expected an integer value as a parameter, and you wished to pass to that procedure a floating point value, you would use INTF to convert the integer to floating point.

Hexadecimal to decimal

Organiser allows you to convert *actual* hexadecimal values to their decimal equivalents, by *prefixing* the hexadecimal number with the **$** symbol. Thus

```
D%=$FF
```

would result in '255' – the decimal equivalent of the hexadecimal value 'FF' – being stored in the integer variable D%.

*The **$** prefix cannot be used with a variable, only actual hexadecimal values.* Nor can it be embodied at the start of a hexadecimal string variable, for use with the VAL instruction. Thus I%=VAL("$FF") is not a valid instruction: it will produce a **STR TO NUM ERR**.

3.11

BUILT-IN DATA AVAILABLE ON DEMAND

OPL words covered
DATIM$,DAY,FREE,HOUR,MINUTE, MONTH, PI
RANDOMIZE, SECOND, YEAR

Time-related information

As you know, Organiser has a built-in clock which, provided it has been set, will keep you up to date about the current time, day of the week, month and year. This information is stored in Organiser and is continually up-dated, even when Organiser is switched off (remember JIM?).

There are a number of OPL words which allow you to access this information for use in your own programs. You may, for example, write a program that keeps a record of car journeys – and wish to tag each entry with the date. You *could* enter the date along with the entry, but it is easier if the program is written so that the date is added automatically.

The OPL words used to access the built in information are *reserved* for OPL's own use: you cannot use them as your own variables. For example, YEAR is an OPL word. You cannot name one of your own variables as 'YEAR' (although you could use 'YEAR%'), and you cannot assign any other value to YEAR.

It is assumed that you have correctly set the system clock: the 'current' time referred to in the following paragraphs means the time *in the system clock* at the moment the instruction is executed. If the system clock is incorrect, then the information derived from the clock, obviously, will also be incorrect.

DATIM$

This gives the current day of the week, date and time – including the seconds. If the entire DATIM$ string is assigned to a string variable, the variable must have been allocated at least 24 memory boxes. However, it is possible to 'pick out' required elements of the string, by using the various string handling instructions (Chapter 3.5). The DATIM$ string is always in the form

SUN 17 AUG 1986 15:33:45

so if you wished to select simply the *date* part of the string – without the name of the day – you would use an instruction such as

```
D$=MID$(DATIM$,5,11)
```

This would pick out just the 'date' part of 'DATIM$' (starting from

the 5th character and taking in 11 characters altogether), and store it in 'D$' in the form "17 AUG 1986". Remember that 'DATIM$' gives the *current* date and time information, and is continually being updated – every second.

YEAR

This gives an integer value representing the *current* year. A typical instruction would be

```
Y%=YEAR
```

MONTH

This gives an integer value representing the current month. Thus, if the current month is August, then

```
M%=MONTH
```

would store the value '8' in M%.

DAY

This gives an integer value corresponding to the current date of the month. A typical instruction would be

```
D%=DAY
```

HOUR

This gives an integer value corresponding to the current hour of the day, using the 24-hour clock. A typical instruction would be

```
H%=HOUR
```

MINUTE

This gives an integer value corresponding to the current minute. A typical instruction would be

```
M%=MINUTE
```

SECOND

This gives an integer value related to the current second. You could use the MINUTE and SECOND instructions to create a simple timer. For example:

```
TIMER:
LOCAL M%,S%
PRINT "MINUTES-59 Max"
```

```
INPUT M%
M%=-M%*(M%<60)-59*(M%>59)
CLS
PRINT "SECONDS-59 Max"
INPUT S%
S%=S%*(S%<60)-59*(S%>59)
CLS
PRINT "ANY KEY TO START"
GET
CLS
M%=M%+MINUTE-1*((S%+SECOND)>59)
M%=M%+60*(M%>59)
S%=S%+SECOND
S%=S%+60*(S%>59)
PRINT "STOPTIME-";M%,S%
WHILE (S%<>SECOND) OR (M%<>MINUTE)
ENDWH
PRINT "TIME UP"
BEEP 800,800
```

Program 3.11.1 'TIMER' A simple timing procedure

Some lines in this procedure may need explaining. Following the inputs of the number of Minutes and Seconds to be timed, a check is made to ensure that a value less than 59 (in each instance) has been entered – and if not, the corresponding delay is made equal to 59. Take the line

```
S%=-S%*(S%<60)-59*(S%>59)
```

for example. If a value less than 59 has been entered into S%, then 'S%<60)' is *true*, returning '–1' which when multiplied by '-S%' gives S% – the original value. The second part opf the line will equate to zero since it is *not true* that S% is greater than 59. If it *were* true, then the situation would be reversed – the first part of the line would result in zero, and the second part would result in a value of '+59'. Thus S% is never allowed to have a value greater than 59, whatever is entered.

Once a key is pressed to start the timing, the required Minute and Seconds *delay* is added to the *current* time: if this results in a value *greater* than 59, then 60 is deducted from the result. If it is necessary to adjust the *seconds* value this way, then '1' must be added to the minute value, to get the correct 'stop time'.

A WHILE/ENDWH loop is used to keep Organiser 'spinning around' while waiting for the system clock to catch up with the required 'stop time'. You could add an alarm signal at the end instead of the simple BEEP – this is covered in the next Chapter.

How much memory is left?

Organiser will let you see exactly how many RAM memory boxes you have left. The instruction is 'FREE', and it can be used in the CALCulator mode by simply entering **FREE**. (You could write a procedure to print the amount of memory space still available, but there seems little point).

The use of this instruction will give you a more accurate picture of the space available in Organiser than the INFO option on the main Menu, which gives only the percentages of space used and free.

The FREE instruction cannot be used to ascertain how much space is left on a Datapak: for this, there is another OPL word 'SPACE', which is covered in the Chapters on File-Handling.

Mathematical values

PI

Organiser has the value of the mathematical constant, 'pi', built in. It can be accessed by 'PI', and used in an expression:

```
VAR=4*PI/180
```

As with other OPL reserved words, you cannot change the value of PI, or declare PI as a variable in one of your own programs.

RANDOMIZE

Organiser allows you to generate a *constant* random number. We have already seen that the instruction 'RND' will generate a decimal number between 0 and 1, at random. There is also an instruction, 'RANDOMIZE', which enables the *same* random number sequence to be generated every time. You may want to do this, for example, if testing out a program that uses RND, so that you can repeat the program using the same sequence of random numbers.

RANDOMIZE must be followed by a value (any value you choose). That value is then used as the 'start' point for generating random numbers. Since the same start point would be used each time the program is run, the same sequence of random numbers will occur.

Mathematical functions

A full range of mathematical functions is built into Organiser – for evaluating square roots, sines, cosines and so on. The complete list of these is given in Chapter 2.6, under the heading *The built-in functions.*

3.12

THE SOUND OF MUSIC

OPL word covered
BEEP

Organiser's beep show

You will undoubtedly have heard some of the sounds that your Organiser makes – for alarm calls, perhaps. There is an OPL word – BEEP – which enables you to create sounds for use in your own programs. This Chapter is devoted entirely to the use of that word.

BEEP must be followed by two values, separated by a comma. Thus

```
BEEP length%,note%
```

The *first* value determines how many *milliseconds* the sound will last.

The *frequency* of the sound produced is determined by the second value, as follows

Frequency = 921600/(78+2*note%) Hz

This can be re-written as

note% = (460800/Frequency)–39

to enable us to determine what value we should give to the value 'note%' for a given frequency.

Thus, to play a note that has a frequency of 800 Hz for a quarter of a second (250 *milliseconds*), you would write

```
BEEP 250,(921600/800)-39
```

You will no doubt wish to hear the sounds that can be produced. A very simple *function* type of procedure will allow you to test the tones that can be created using 'BEEP', with the Organiser set to its CALCulator mode. We will write the procedure so that you enter the *frequency* of the note that you want:

```
BP:(L,F)
BEEP L,(921600/F)-39
```

Program 3.12.1 'BP' BEEP function for frequency inputs

Having entered the procedure, set Organiser to the CALC mode,

then type in **BP:(250,800)** and press **EXE**. You will hear a short tone at a frequency of 800Hz. Experiment by changing the length and frequency of the note, so that you can hear the spectrum of tonal sounds available from Organiser.

Keyboard music-maker

As a programming exercise, we will use 'BEEP' to turn your Organiser into a musical (?) instrument. Of sorts.

The Internationally accepted frequency for the note of 'A' is 440Hz. To find the frequency of a note one octave higher, the frequency is doubled. Using this scant information, a formula can be devised to determine the frequency of any note in the scale.

If we say that note 'A' is the start of the scale, then 'A sharp' is the second note, B is the third and so on. Calling this the note *number*, the formula is

note frequency = 440*2(note number/12)**

Thus, for the note of 'C' – the fourth note from 'A' – the frequency is 440*2**(4/12), or 554.365.

Using this principle, Organiser's keyboard can be converted into a musical keyboard. There is no easy way to identify the 'black notes' – and so, for the purposes of demonstration, we will simply code each letter from 'A' to 'Z' as consecutive tones.

There are two ways to approach this. The first way is to calculate the frequency – and then the value required for the BEEP instruction – each time a key is pressed. This would result in a delay between pressing the key and the note sounding – whilst the calculations are made.

The alternative is to calculate all the parameters required for BEEP *first* and to store them in an array. Then, when a note is called for, it will simply be a matter of selecting the appropriate element in the array. This will result in a considerably faster processing time – and the keyboard will more faithfully obey your keypress commands.

Ideally, each note would sound only whilst a key is being held down. However, when a key is pressed there is a short delay before it auto-repeats quickly. This means there would be a gap in the note. The alternative is to arrange for the note to play continuously until another key is pressed – and use the **SPACE** key to switch the sound off.

Since Organiser's keyboard does not resemble a piano keyboard, it will be useful to know which note is being played when a key is pressed. This information can be set up in a string array, similar to the note% information. The task can be made easier by setting up the first 12 notes, then getting Organiser to assign the rest in a DO/UNTIL loop.

Finally, as all the alphabetic keys are being used to produce notes, we will arrange that any of the control keys – **EXE**, **DEL** and so on – will exit the program altogther. **SPACE**, remember, is going to be used to stop a note from playing.

The variables used in the program are as follows:

K%	holds the ASCII value of the pressed key.
L%	holds the value of the last key pressed, until the **SPACE** key is pressed.
F	holds the calculated Frequency for the note.
N(26)	holds the BEEP information for all the keyboard.
N$(26,2)	holds the name of the note for each key.

```
PLAY:
LOCAL K%,L%,F,N(26),N$(26,2)
N$(1)="A "                :REM Set up 1st 12 notes.
N$(2)="Bb"                :REM Note  space after
N$(3)="B "                :REM single letter notes.
N$(4)="C "
N$(5)="Db"
N$(6)="D "
N$(7)="Eb"
N$(8)="E "
N$(9)="F "
N$(10)="Gb"
N$(11)="G "
N$(12)="Ab"
L%=13                     :REM Organiser does rest.
DO
  N$(L%)=N$(L%-12)
  L%=L%+1
UNTIL L%=27
K%=1
PRINT "  INITIALISING"
DO
  F=440*2**(K%/12.)       :REM Note Decimal point
  N(K%)=(912600/F)-39     :REM Set up note array
  K%=K%+1
UNTIL K%=27               :REM All done
CLS
PRINT "PLAY AWAY..."
PRINT "NOTE="
KSTAT 1                   :REM Set for capitals
L%=0                      :REM No last note
ST::
K%=KEY                    :REM Get a keypress
IF (K%=0) AND (L%<>0)     :REM Note being played
  GOTO PL::
ELSEIF K%=0               :REM No note being played
  GOTO ST::               :REM Go get keypress
```



```
ELSEIF K%=32              :REM SPACE=stop sounds
  L%=0                    :REM No last note
  AT 6,2
  PRINT "  "              :REM Clear note display
  GOTO ST::               :REM Go get keypress
ELSEIF K%<65              :REM = control key
  RETURN                  :REM so exit program
ELSE L%=K%                :REM Must be a note key
  AT 6,2                  :REM so L%=last note
  PRINT N$(L%-64)         :REM Display note
ENDIF
PL::
BEEP 25,N(L%-64)          :REM Play note quickly
GOTO ST::                 :REM Go for a keypress
```

Program 3.12.2 'PLAY' Making Organiser a keyboard instrument.

As mentioned for previous programs, you do not need to enter the 'REMarks' (or the space and colon preceding them), nor do you need to enter the indents preceding the instructions. They're there to help you understand how the program operates.

The first part of this program sets up the arrays containing the 'note%' information, and the information for the display of the note. It takes a few seconds to complete, when running – hence the display "INITIALISING", so that you don't think something has gone wrong.

Then comes the 'loop' to get the key press information and to determine the action to be taken. Once an alphabetic key has been pressed, the variable 'L%' is set up to hold the ASCII value of the key (65 to 90 for A to Z), and the note name is displayed on the screen by selecting the appropriate element in the 'N$()' array. ('65' is the *first* note, so by deducting 64 from the ASCII value, we get the *number* of the note – and hence its position in the array). The note is then played for a fraction of a second, and a jump made back to the key input part of the program.

Now, until another key is pressed, K% will be equal to zero and L% will be holding the ASCII value of the key just pressed. Consequently the first test 'IF (K%=0) AND (L%<>0)' will be true, and Organiser will jump straight down to the 'PL::' label to play the
note again. And so the process repeats.

If another alphabetic key is pressed, the value held by L% is updated and the note is changed. If **SPACE** is pressed (ASCII value of 32), L% is made zero. The displayed note is cleared, and a jump made back to the start label 'ST::' – so Organiser will not make a sound (some will bless you for that). If a control key is pressed – **EXE** perhaps – the program ends. (You may be blessed even more for that!).

This program can be modified in a number of ways: for example,

the *length* of the note (25) can be changed to produce different 'vibrato' sounds. If it is made *too* long, however, Organiser will not respond to key-presses as effectively.

Sound effects

As well as providing specific notes, the BEEP instruction can be used to provide a variety of sound effects. This is demonstrated with three procedures. The first shows how the sound of a ringing 'trimphone' can be simulated. The second shows how a simple two-tone alarm signal can be created, while the third will allow you to produce the 'space age' sounds so common with modern electronic devices. (Well ... why should your Organiser be left out?)

These procedures should be fairly self explanatory: it is left to you to experiment further, should you so desire.

```
PHONE:
LOCAL L%,C%,D%
DO
  D%=KEY
  L%=1
  DO
    C%=1
    DO
      BEEP 30,700
      PAUSE 1
      C%=C%+1
    UNTIL C%>12
    L%=L%+1
    PAUSE 4
  UNTIL L%>2
  PAUSE 15
UNTIL D%<>0
```

Program 3.12.3 'PHONE' Telephone sound effect

The 'phone' will stop ringing shortly after any key is pressed. In this procedure, the BEEP instruction produces a constant sound. The rest of the procedure arranges for the sound to be produced at approximately the same rate as a telephone: you can adjust the timing quite easily, by altering the values of the variables in the 'UNTIL' instructions, and by changing the values after the 'PAUSE' instructions.

The following procedure shows how simple it is to produce an 'alarm' type signal.


```
ALM:
LOCAL C%
C%=1
DO
  BEEP 50,800
  BEEP 50,1600
  C%=C%+1
UNTIL C%=20
```

Program 3.12.4 'ALM' Alarm type signal

In the next procedure, the frequency of the note produced is changed within 'DO/UNTIL' loops, by using the loop 'counter' in some way. The possibilities using this technique are virtually limitless.

```
SFX:
LOCAL I%,K%
K%=2000
DO
  K%=K%-200
  I%=20
  DO
    BEEP 5,I%+K%
    I%=I%+(I%*.5)
  UNTIL I%>2000
UNTIL K%<200
```

Program 3.12.5 'SFX' Sound effects sampler.

3.13

FILES – CREATING AND OPENING

OPL words covered
CREATE, EXIST, OPEN

Introduction to files

We have already seen, in Chapter 2.4, how Organiser lets you keep records in its 'built-in' filing system. This is fine for fairly simple requirements, but it does not provide for the information contained in the records to be *processed*. For example, while you can use it to keep a list of club members, their membership fees, and whether or not they have paid those fees, what you cannot do (from the built-in main Menu options) is get Organiser to tell you the *total* sum of fees paid – or fees due. You would have to examine each record in turn, and make the calculation manually.

By writing a suitable program, you can create your *own* filing system and get Organiser to perform any task that you wish. You can also access the Organiser's built-in file from your own program (it is called 'MAIN' remember).

OPL has a large number of words devoted entirely to file handling. These words enable you to create files, add records, change records, delete records, search records for specific information, manipulate the information, use it in calculations – and so on. Indeed, the file handling capabilities of OPL are extremely powerful – more powerful than can be found on most home computer systems, and the equal of many office systems.

This means that you can use Organiser not just to make *calculations* – as we have seen in earlier Chapters – but to process data. A 'payroll' program for a small company, or a stock-handling system, for example, are well within Organiser's capabilities.

What goes into a file

In Chapter 2.4, we saw that a file is like a card index box containing record cards. For the built-in filing system accessible from the main Menu, that is as far as it goes.

When you create your own files, the information contained on each record 'card' follows the same format. If you were to write information on actual cards, you would probably arrange things so that each *area* of the card carries the information in a regular way. If names are involved, for example, you may decide these should always appear on the top line. You may reserve the next few lines for the address, the bottom right hand corner for some kind of

classification regarding the named person – and so on. Each of these areas is called a *field*.

Thus, a file consists of *records*, and a record consists of *fields*. When you create a file on Organiser, you specify the fields that you want, and *name* them so that they can be identified – as 'variables'. (The records in Organiser's built-in file 'MAIN' has only one field).

For example, if you were creating a file for employees, you would probably have record fields for their *name*, *address*, *tax-code*, *salary*, *pay-to-date*, and so on. Alternatively, if you were creating a stock and price list file, you would probably have record fields for the *item description*, its *stock-number*, its *price*, the *current-stock*, and perhaps the *minimum stock-holding* (so that you could quickly ascertain which items need re-ordering).

It could well be that you would want more than one file of the same type – perhaps one stock/price list for perishable goods, and another for non-perishable goods. No problem. Organiser II lets you have as many as 110 files (if space permits) at any one location (RAM or a Datapak) – and, more importantly, *lets you work on up to four of them at the same time*. You can transfer information from one to another, or perform whatever operation is necessary, by simply writing an appropriate program.

The important point to remember is that, whereas programs store *instructions*, files hold information or *data* – and the data must be stored in an orderly fashion.

The file handling process

When writing a program to create and handle a file (or files), the program sequence is as follows:

a) If a file does not yet exist, *create* it.
   If a file already exists, *open* it.
   Files can be opened (and *closed*) at will throughout a program, but a file must be *open* in order to 'work' on it.
b) Perform the necessary file handling operations. This will usually involve selecting from a choice of options (using the 'MENU' technique): typical options would be to add or delete a record, to change 'field' information, make a calculation based on information in a specific field or fields, find a particular item of information, transfer or manipulate information from one file to another, and so on. You can even delete *unopened* files, copy them (to or from a Datapak), or change their name.
c) Close the file (or files)

Copying files from one place in memory to another, changing file names and deleting files is generally referred to as *file management*. If you have a number of files, you may well choose to have a file

management program quite seperate from the file *handling* program. This management program will enable you to perform the copying, name-changing and deleting operations on *any* files you have saved, irrespective of the programs that use those files.

Creating a file

The OPL word that enables you to create a file is, not surprisingly, CREATE. The complete instruction takes the form

```
CREATE "Loc:name",logfile,field1,field2,field3...field16
```

That's quite an instruction! It is important that you understand what the parameters completing the instruction mean and how to use them, so let us examine each in turn.

"Loc:name"

This is a *string* that defines first *where* the file will be created, followed by a colon, and then the name that it will be given. If you do not have Datapaks fitted, then the 'Loc' will be "A:". This means that your file will be created in Organiser's RAM. If you have Datapaks fitted, then you can create your files on a Datapak by specifying "B:" for the Datapak fitted in the upper slot, and "C:" for the Datapak in the lower slot. Note that it is the *location* of the Datapak that is specified – so be sure the right Datapak is plugged into the appropriate slot before the program is run.

The second part of the string specifies the name that you wish to give the file. This name must obey the same rules as procedure names: it must be no more than eight characters long, can comprise only letters and numbers, and must start with a letter. You *can* include a **$** or a **%** symbol as the last character, if you wish – but there is no *programming* reason to do so.

This parameter can be an *actual* string – between quotation marks, or it can be a string variable, provided of course that the string variable holds the correct information.

Typical file names would be "A:STORES", "B:FRIENDS", "C:CLUBMEM". The file named "STORES" would be saved in

Organiser's RAM, while the files "FRIENDS" and "CLUBMEM" would be saved on the upper and lower Datapaks respectively.

logfile

It has been mentioned that up to four files can be 'open' at a time for working on. Organiser needs to know which of the four files to use for your instructions: you may have *more* files

available – and obviously it could lead to lengthy programming if you had to specify the file *name* each time. This problem is obviated by giving each of (up to) four files an identifying letter – A, B, C or D. When you create or open a file, therefore, you must tell Organiser how you intend to identify it throughout the rest of the program. So if you make the 'logfile' parameter 'A', say, then the associated file will be referred to throughout the program as 'A' – until it is 'closed' and another file opened as 'A'. This means that the program will operate with *whatever* file you choose to open as 'A' – and the program does not have to cater for all the file *names* that you have created.

Field1...Field16

As indicated, each record in your file can have *up to* 16 fields. These fields are given names *just like variables*. Indeed, within your program, you can refer to the fields in exactly the same way as you refer to variables, with one difference: they must be preceded by the *logfile* letter and a full point. Thus if you name a field as 'STOCK$' in a file opened as logfile 'A', then you can use that field name as a variable, *without declaring it*, by writing 'A.STOCK$'. The information contained in a variable is *not* entered into a record until you give the correct instruction. Thus, the field names can be used as *global* variables.

Just like variables, the field names must reflect the nature of the information to be stored in that particular part of the record: if the information is to be stored as characters, the name must terminate with a **$** symbol, and if it is integer numbers, the name must terminate with a **%** symbol. Fields for floating point numbers do not require an identifier.

The maximum length for a field name, just like a variable, is eight characters, including the identifier, and it must obey the same rules.

Thus, if you wished to open a file in Organiser's RAM with the name "STOCK", making it the first (and perhaps only) file to be used in the subsequent program, and you wanted each record to have fields for "ITEM$", "PRICE","STOCK%", and "MINSTOK%", then the instruction would look like this:

```
CREATE "A:STOCK",A,ITEM$,PRICE,STOCK%,MINSTOK%
```

Alternatively, you may wish to have the program such that you can create *new* files at various times, in which case the section of the program could look like this:

```
PRINT "FILE NAME="
INPUT FN$
CREATE FN$,A,ITEM$,PRICE,STOCK%,MINSTOK%
```

The information for 'FN$' must follow the correct form for the file-name, of course. For the above example, you would type in at the keyboard **A:STOCK** *without* quotation marks.

Organiser will not let you create a file with the same name as another file *at the same location*. You can, however, create a file with the same name as another, provided it is at a different location – although this could be a dangerous exercise. It is better practice to keep each file individually named, wherever it is saved, unless you have specific reason for doing otherwise.

When a file is *created*, it is automatically *opened* for use. What you have done is to create a named *filing box*, and defined the names of the *fields* that will appear on each of the individual *record cards* that will ultimately go into the box. You have also told Organiser, through the *logfile* letter, how you are going to refer to that file in the file-handling instructions of your program.

Organiser also needs to be able to identify the *end* of the area in RAM it will be reserving for the field information. Consequently, *immediately* after *creating* a file it is necessary to assign a value to the *last named* field, and add the blank record to the file. Naturally you will not want the first record to be a blank – and so it should be erased immediately. The two OPL words concerned are 'APPEND' and 'ERASE', which are dealt with in the following Chapters. If the *logfile* is 'A' and the last field you have named is 'MINSTOK%', then after *creating* the file you would need the instructions

```
A.MINSTOK%=0
APPEND
ERASE
```

Although you can work on up to four files at a time, only *one* is the *current* file – and that is always the last one worked on. Thus, if you create a file with the logfile identification of 'A', then that is the file that will be used for access until another file is made *current* (by other instructions).

Opening a file

Obviously you need only create a file *once*. Once it has been created, thereafter whenever you wish to use the file in a program, you need only *open* it. The OPL word is OPEN, and it must be followed by the same set of parameters as CREATE. You would of course use the appropriate *logfile* letter for the ensuing program. You can also give the fields different names to those used when it was created – but they *must* be of the same *type* (string$, integer% or floating point). However, it will make your programs easier for you to understand if you keep to the same names for the fields wherever possible.

When writing a program entailing the use of a file, you must cater for the fact that the very first time it is run there will be no file in existence. You can do this in one of at least two ways. You can start the program off with a Menu, of which one of the options is 'CREATE'. If this option is selected, you would jump to instructions that create the necessary file.

Alternatively, you can use the OPL word 'EXIST' to test whether the file exists, and act accordingly if it doesn't.

EXIST must be followed by a string of the file location and name, in brackets. Thus:

```
EXIST ("A:STOCK")
```

The name in brackets can be an actual string, within quotes, or a string variable. On meeting this instruction Organiser searches the *named location* for the *file-name*. If it finds the file at the named location, it returns 'true'. If it doesn't, it returns 'not true'. Thus, you could have a section of program:

```
IF EXIST ("A:STOCK")
  OPEN "A:STOCK",A,ITEM$,STOCK%,PRICE,MINSTOK%
ELSE CREATE "A:STOCK",A,ITEM$,STOCK%,PRICE,MINSTOK%
  A.MINSTOK%=0
  APPEND
  ERASE
ENDIF
```

In this instance, if you have a file called 'STOCK' on one of the *Datapaks*, Organiser would not find it in RAM ("A:"), and so it would create another file called 'STOCK', in RAM. If creating or opening a file on a Datapak, you must be very careful to properly locate your Datapaks before running the program.

You can also use the EXIST instruction when creating a new file using the Menu technique, to save Organiser breaking out of the program to report an error if the file-name you have chosen already exists. A part of such a program may look like this

```
GETNAME::
PRINT "FILE NAME="
INPUT FN$
IF EXIST (FN$)
  PRINT "THAT NAME EXISTS"
  PAUSE -60
  GOTO GETNAME::
ELSE CREATE FN$,A,ITEM$,STOCK%,PRICE,MINSTOK%
  A.MINSTOK%=0
  APPEND
  ERASE
ENDIF
```

You should also arrange that your program tests that a *valid* location and file-name is entered for the variable. There are a number of ways to do this. Here's one:

```
INPUT FN$
T$=UPPER$(LEFT$(FN$,1))
IF (T$<"A") OR (T$>"C")
  GOTO GETNAME::
ELSEIF MID$(FN$,2,1)<>":"
  GOTO GETNAME::
ELSEIF EXIST (FN$)
and so on
```

The first IF instruction tests that the first letter is either A, B or C. The entry may have been a lower case letter – which Organiser will accept – and so the entry is first converted to capitals using the UPPER$ instruction. This is followed by a test that the *second* character in the string is a colon. If these tests prove satisfactory, processing can continue. Otherwise, a jump is made back to get the file-name again. You could, of course, incorporate a message that the file name was of the incorrect form.

There is another way to prevent an error from causing Organiser to break out of the program: this is discussed in Chapter 3.17. The important thing to remember, always, is to make your programs as error-free as possible when running – and to allow for the fact that a user may make errors when entering information.

As mentioned earlier, you can open up to four files at a time, although each must have a different *logfile* letter from A to D. The files can be located anywhere in memory – in RAM or on Datapaks.

3.14

FILES – WRITING AND CHANGING RECORDS

OPL words covered
APPEND, EDIT, RECSIZE, UPDATE, USE

Putting information on record

Two OPL words enable you to enter information into the fields of a record. These are 'APPEND' and 'UPDATE'. APPEND is used to add a record, and UPDATE is used to change a record. Another OPL word, 'EDIT', allows you to change the information held in a *string* type of field.

The field names, remember, can be used in the same way as variables. It is the information contained in these variables that gets written into the appropriate parts of the file record.

Adding a new record

APPEND adds the information in *all* the current field-name variables to a *new* record in the file. This is the instruction to use, therefore, to add records to the file. The new record is *always* added at the *end* of the file – you cannot add records in the middle: this is not a problem – the records do not have to be in any specific order for you to locate and use them, as would be the case with a physical card-index box system.

If more than one file has been opened, the APPEND instruction applies to the file *currently* in use – that is, the last one actually worked on – unless otherwise instructed by the the OPL word USE (discussed in more detail later on). If you have opened three files with similar field-names, the names related to each specific file will be identified by the *logfile* letter – so you could have variables 'A.STOCK$', 'B.STOCK$', 'C.STOCK$', and so on. If *logfile* 'A' is the one currently in use, then APPEND will write a new record into the file designated as *logfile* 'A', **using all the 'A.' prefixed variables**.

The procedure is to assign the required information to the variables, and then give the instruction APPEND – which is complete in itself.

To give an example, suppose we have opened only *one* file, as *logfile* 'A', and it has fields of ITEM$, STOCK%, DAY% and MONTH%. We wish to add a new record to this file. We want to enter the ITEM name and the current STOCK from the keyboard

and we want to use the built in clock and date information to add the current date.

The relevant part of the program could look like this:

```
PRINT "ITEM NAME-"
INPUT A.ITEM$
PRINT "CURRENT STOCK-"
INPUT A.STOCK%
A.DAY%=DAY
A.MONTH%=MONTH
APPEND
```

On the instruction APPEND, the information contained in all the 'A' variables is written to the file as a new record. By looping back, another record can be added to the same file, using the same instructions: you would, of course, need a way to exit the loop. This could be done by following the APPEND instruction with a Menu, or by testing the first input (for ITEM$) for a particular character or series of characters. An IF test following 'INPUT A.ITEM$' could, for example, check whether "NO" had been entered, and if so, would arrange for processing to jump away from the 'adding records' part of the program.

When you are using your file, Organiser maintains 'pointers' to the current *file* and the current *record*. When you add a record using APPEND, it is added to the *current file*, and the added record then becomes the *current record*.

As we shall see, there are ways to move or position the 'pointers' to any opened file, and to any record in that file. Whenever this is done, the *field-name variables hold the information contained in the record pointed to* – the current record. This is quite an important point to remember: unlike 'ordinary' variables, the information contained in field-name variables will change as the record is changed. So whilst *you* can assign information to the variables, if a new record is selected without that information being saved, the information you assigned will be lost.

Changing a complete record

This brings us to the second way to put information on record – UPDATE. This *replaces* the information in the current file record with the information currently held in *all* the field-name variables.

What actually happens is the entire record is written again *at the end*, and the original record is deleted automatically. In terms of the card index box, when you alter a record you take out the card, make the alteration, and replace it at the *back* of the box. As mentioned before, the order of the cards in the box is quite immaterial to the operations that you may wish to perform: the cards are in fact in a 'historic' order – the order that you have

entered them or changed them. It is important to remember this – if a particular record happens to be the third (say) on the file, its position will change if you perform an UPDATE on any of the first *three* records. If a file is on a Datapak, then of course the original record is not *erased*, but is *locked up* so that it can no longer be accessed. Thus, changing records on a Datapak will gradually consume the space available.

Why should the record be re-written afresh at the end? Because files hold their record information in a tight sequence: when you make a change, the new record may then contain more characters than the original – and there would be no space available to put them in the original record. Hence it is re-written. With Datapaks, of course, you could not 'write' a new record over the top of an old one anyway.

We shall see, in the next Chapter, how any record can be selected in the current file. When selected, the record becomes the *current* record, and all the field-name variables will hold the information contained in that record. You can assign a new value (or string) to any or all of these variables, and then write *all* of them back to the record, using UPDATE.

The process can be demonstrated by a small part of a procedure. We will assume that we have a *current* file as *logfile* 'B.', record field-names of 'ITEM$, STOCK%, PRICE, MINSTOK%, DAY%, MONTH%', and that we have selected a particular record we wish to change (so that the record is *current*). We want to cater for changing one or more of the record information items 'ITEM$, STOCK%, PRICE, MINSTOK%' – and we always want the current date to be added to the record, so that we will know when the record was last changed. The relevant part of the procedure could look something like this:

```
WHILE C%<>5
C%=MENU("ITEM,STOCK%,PRICE,MINSTOK%,SAVE")
IF C%=1
  PRINT "ITEM BECOMES"
  INPUT B.ITEM$
ELSEIF C%=2
  PRINT "NEW STOCK-"
  INPUT B.STOCK%
ELSEIF C%=3
  PRINT "NEW PRICE-"
  INPUT B.PRICE
ELSEIF C%=4
  PRINT "NEW MIN-STOCK-"
  INPUT B.MINSTOK%
ELSEIF C%=5
  B.DAY%=DAY
```

```
    B.MONTH%=MONTH
    UPDATE
  ENDIF
  ENDWH
```

A few words of explanation on how this section of a procedure operates. On reaching the instruction 'WHILE C%<>5', C% must of course not be equal to 5 – otherwise a jump will be made straight to 'ENDWH'. As long as C% is not equal to 5, the next instruction is obeyed – to display a Menu on the screen. When an option to change information is selected, C% is used in IF tests to direct Organiser to display the appropriate prompt and get the required information into the field-name variable. The relevant IF instruction having been obeyed, Organiser jumps to 'ENDIF', 'ENDWH' and back to 'WHILE C%<>5'. Since C% will still not be equal to 5, the routine is repeated – and the information contained in the field-name variables can be changed ad nauseum.

When 'SAVE' is selected, the DAY% and MONTH% field-name variables are assigned the current date values, and *all* the variables are written back to a new record in the file (the original record being 'erased'). This time, the 'WHILE C%<>5' instruction will prove to be *not true* (C% *is* equal to 5), and Organiser will jump straight to 'ENDWH' to continue with the rest of the program.

See what you are changing

The procedure given in the previous paragraphs doesn't show you the *currently* held information in the relevant variable. It merely prompts you to enter the new information. You could display the current information on the screen by using a PRINT instruction. To see what the current item is, for example, you could add PRINT B.ITEM$ before or perhaps instead of the prompt.

However, with only two lines of screen, you could loose the display by the time you get to the actual INPUT instruction. There is an alternative – the EDIT instruction.

EDIT operates only on *string variables*, the full instruction taking the form EDIT string$.

When Organiser reaches an EDIT instruction, it displays the string on the screen. The string can then be *edited* by using the cursor keys and the **DEL** key to locate and remove the unwanted characters, and by using the character keys to enter the new information. Pressing **CLEAR/ON** clears the entire entry, and allows you to type in information afresh.

The EDIT string is displayed on one line: if it is longer than 16 characters, then the cursor keys can be used to access the 'hidden' parts of the string.

Thus, you have a way to display existing record information on the screen and to *edit* that information, rather than having to enter it all over again for perhaps just one or two changes.

Yes, 'EDIT' works only on strings.

So what value is it for *integer* and *floating point* variables?

None, *directly*. But, you will recall from Chapter 3.10, you can convert numeric variables to strings – and then convert them back again. So if you wanted to use the EDIT instruction on numeric variables, you can. Let us take a small section of the previous example, and show how the information contained in the 'STOCK%' field can be viewed and changed. (Variables other than field-name variables must be declared, of course).

```
STOCK$=NUM$(B.STOCK%,5)
KSTAT 3
EDIT STOCK$
B.STOCK%=VAL(STOCK$)
```

In this example, the 'NUM$' conversion instruction is used. This is adequate for B.STOCK% values up to 32767, the maximum that can be contained in an integer variable. You can of course use any of the number-to-string conversions available.

Using this system, you will have to be careful that *numbers* are entered during the 'EDIT', otherwise a **STR TO NUM ERR** will occur when the string is converted back to the numeric variable 'B.STOCK%'. Setting the keyboard for numeric inputs ('KSTAT 3' or 'KSTAT 4') helps – but will not stop other *keyboard* characters such as < or + being entered. Ways to 'trap' errors of this kind are discussed in Chapter 3.17.

The string is re-converted back to the integer variable by the instruction 'B.STOCK%=VAL(STOCK$)', ready for the 'UPDATE' instruction.

While EDIT is of most use during file-handling operations, it can also be used in other types of program, and can be a useful 'tool' when *de-bugging* programs. (See Chapter 3.17).

Maximum record size

The number of characters that can be contained in each record is 254. This is normally quite large enough, even for records that have a full complement of fields, and should not cause a problem. You can gauge what 254 characters looks like by counting the number of letters, spaces, commas and so on in your own name and address – you will usually find that it is around the 50 mark. This means there will still be around 200 characters left for other information.

However, if you do have fairly long records, it can be useful to know whether you are approaching the limit, so that you can perhaps use abbreviated information in order to contain everything you want in the record. There is an OPL word, 'RECSIZE', which gives the number of characters used in the *current* record. The full instruction takes the form

```
R%=RECSIZE
```

This instruction can be used to display a warning should an input to a file cause the number of characters to exceed the limit. Assuming that a field 'A.DATA$' is being updated, a typical routine could look like this:

```
label::
INPUT A$
A.DATA$=""
IF RECSIZE+LEN(A$)>254
  PRINT "RECORD TOO LONG"
  GET
  GOTO label::
ELSE A.DATA$=A$
ENDIF
```

A routine such as this could be included immediately before an 'APPEND' or 'UPDATE' instruction, sending the processing back to where the information is being assigned to the field-name variables, for re-entering. However, it must be pointed out again that such a measure is not necessary for most types of record. It is only if lengthy records are being used that a check needs to be made, to prevent Organiser from stopping the program to report an error because the record size is too large.

Selecting which file to use

It has been stated that you can have up to four files open at any one time, and that only one of these files is *current* at a time. The *current* file is the one on which Organiser can perform its various operations. Put another way, a file *must* be current if Organiser is to perform any operations on it.

The current file is the one that has just been worked on – for an APPEND or UPDATE operation, perhaps. If four files are opened (or created), the current file is the *last* of the four.

Suppose you want to perform an operation on one of the *other* files? That is when the OPL instruction 'USE' is used.

USE must be followed by a *logfile* letter corresponding to a file that has been *opened*. If an incorrect letter is used, or a file has not been opened with the *logfile* letter, then an error will occur.

While files are opened, *all* of the field-name variables are available for use *all* of the time. It is only when *file operations* are performed – such as writing information to a record – that the 'USE' instruction needs to be used if the required file is not the current file.

You can refer to the fields of different files without affecting the current file. For example, you may wish to perform an operation involving the fields of two files:


```
A.TOTAL=B.COST+C.COST
```

If *logfile* 'B' is the current file, it will still be the current file after this instruction line. So if you wished to save the record involving the field 'A.TOTAL' – which is a *logfile* 'A' record – you would write

```
USE A
UPDATE
```

The current file would then be 'A'.

Note that, in the above example, the field information will be for the same record *number* in each file. Thus, if record 23 (say) had been selected, then 'A.COST', 'B.COST', and 'C.COST' would each relate to record 23 of the respective files.

3.15

FILES – HANDLING RECORDS

OPL words covered
BACK, CLOSE, COUNT, DISP, EOF, ERASE
FIND, FIRST, LAST, NEXT, POS, POSITION

Which record do you want?

So far we have seen how to create and open files, and how to write new records or change existing records in those files. It has also been mentioned that when new records are added to a file, they are added at the *end*. Also, changed records are re-written at the end, and the original record is deleted from the file. This doesn't matter, because Organiser allows you to find any record you want very quickly indeed.

There are three basic ways to locate a record. You can step through each record in turn until you find the one you want. You can search through the records to find a specific item of information. Or, if you know the position of the record in the file, you can go directly to it. We shall discuss each of these methods in turn, then examine the ways you can inspect the information contained within a record.

Step by step

At the moment, remember, we are simply discussing how to make a specific record the *current* record, not how to display that record. OPL has five words to help you to step through your records one at a time. These words are 'FIRST', 'LAST', 'NEXT', 'BACK', and 'EOF' (short for 'End Of File'). The first four words are complete instructions in themselves. The last word, 'EOF', is a *flag*. (Don't panic! The concept of a *flag* is very easy to understand).

FIRST

This sets the record pointer to the first record in the current file, making it the current record. FIRST can be used at any time within the program, to ensure that you are set to the beginning of the file. Some operations (such as 'FIND'), work from the current record to the end of the file, and so if Organiser were pointing to a record in the middle, earlier records would be missed out.

NEXT

As the name implies, this OPL instruction tells Organiser to make the *next* record in the current file the current record. If there is not another record in the file when a NEXT instruction is met, Organiser says to itself "There are no more records, so I'll run a *flag* up the flagpole, to tell everyone". The
pointer will be 'aimed' at the last record in the file but, because the flag is *set*, all of the field-name variables will contain nothing at all. Organiser does not report an error has occured. It expects you to spot the flag.

BACK

This instruction tells Organiser to point to the *previous* record in the current file, making it the current record. If the instruction is given whilst the first record is current, then nothing happens.

LAST

This is the opposite to FIRST: it sets the record pointer to the last record in the current file, making it the current record.

EOF

This is the 'flag' that Organiser uses to indicate that the end of the file has been passed. If EOF is equal to zero (the *not true* condition), the flag is not 'raised' and the end of the file has not been passed. If EOF is equal to '–1', (the *true* condition), you can consider the flag is up the pole, indicating that the end of the file has been passed. *You* cannot assign a value to EOF.

Thus, EOF can be used to terminate a step-by-step search through the records of a file. A typical routine might be as follows:

```
FIRST
DO
  Examine record/Display record
  Perform whatever operation is necessary
  NEXT
UNTIL EOF
```

At the instruction following 'UNTIL EOF', Organiser will be pointing 'past' the end of the file, and the EOF flag will be set.

Searching for specific information

This is the second way to find a record that you may wish to view or work on. The OPL word is 'FIND', and it operates in the same way as

the FIND on the main Menu. More or less.

FIND, as an OPL instruction, must be followed by a *string* contained within brackets. This can be an *actual* string, or a string variable. When Organiser meets the FIND instruction, it searches through all the records *from the current record to the end of the file* to see if the characters *anywhere in the record* match those in the specified string. If the string is 'null' – that is, has no characters in it at all – then FIND behaves just like 'NEXT', to make the next record the current record.

Note that *all* the fields of a record are searched – not just the string fields. So if you wish to find a particular *value* in a numeric field, you would first turn the value you wish to find into a string, then use the string in the FIND instruction. Even though the value may be contained in an integer or floating point type of field, it will be found (if it exists).

When Organiser finds a match, it stops, and 'returns' the *current* record number. Thus typical instructions could be

```
F%=FIND("JIM")
F%=FIND(SEARCH$)
```

In these instances, the record *number* in the file will be stored in 'F%'. Because Organiser stops when it finds a match, the *found* record will become the *current* record, ready for any operations that you may wish to perform.

If Organiser does *not* find a match, then zero is 'returned', and Organiser will be pointing past the end of the file, with the 'EOF' flag set.

The string you give for the search can be as long or as short as you like – the conditions are just the same as those for the main Menu FIND. A part of a procedure to search through records for every occurence of a specific item of information could look like this:

```
PRINT "SEARCH CLUE="
INPUT S$
FIRST
WHILE FIND(S$)
  Perform necessary operations
  NEXT
ENDWH
PRINT "END OF FILE"
GET
```

Notice that before the search is started, Organiser is set to point to the first record in the file. When the instruction 'WHILE FIND(S$)' is met, Organiser starts the search from the *current* record to find a match. If it finds a match, 'FIND(S$)' will have a value (the record

number) – and so the condition will be *true*. The relevant record will then be the current record – so if another search is made, that same record would be found again. Hence the 'NEXT' instruction – to move on to the next record before continuing the search. *Without* NEXT, *Organiser would go into a perpetual loop should the search be successful.*

Having moved the pointer to the next record, 'ENDWH' sends Organiser back to the 'WHILE' instruction, to repeat the search through the remaining records. If no match is found, 'FIND(S$)' will
be equal to zero – the *not true* condition – and so Organiser jumps to the instruction following 'ENDWH', to report the end of the file has been reached. Remember that, at this point, the EOF flag will have been set.

Using FIND in the way just described means that the actual number of the record – that is, its position in the file – is not 'saved' anywhere *by the procedure*. The instruction to save a record number, remember, would be something like 'F%=FIND(S$)' – where the record number is saved in F%. The procedure *could* be re-written to incorporate this type of instruction, if you wanted to know the record number, but it is not necessary. There is an OPL word that will tell you the number of the current record.

The word is POS. An instruction such as 'P%=POS' would store the current record number in P%. 'PRINT POS' will display the current record number on the screen. To test whether a particular record *number* has been found, an instruction such as 'IF POS=value' can be used. Remember, though, that the record number really relates only to the *historic order* of the records in the file.

Going straight to a record number

If you know the number of a record – its position in the file – you can go straight to it by the OPL instruction 'POSITION', which must be followed by the required number. Thus to make the third record in the file the current record, you would use the instruction 'POSITION 3'. (Note that the equals sign is not used).

If you choose a record number greater than the number of records in the file, you will in effect point past the end of the file, with the EOF flag set.

How many records are there?

You may wish to know how many records there are in the file. If your file contains club members, for example, you may wish to know how many club members you have. You *could* write a procedure to count the records. This is unecessary, however, since there is an OPL word – 'COUNT' – to give you the answer straight away.

COUNT returns the number of records in the current file (which must, of course, be *opened*). As with other words of this type, typical instructions are 'C%=COUNT', and 'PRINT COUNT'.

Viewing your records

There are many ways that you can display current record information on the screen. To simply display the information in one of the record fields, you can use 'PRINT' – followed by the field-name as a variable. Thus, to display a field named 'ITEM$', in a file opened as *logfile* 'A', you would simply write 'PRINT A.ITEM$'. The *logfile* does *not* have to be the current file. However, the 'ITEM$' information displayed will be for the *current* record number.

The snag with using 'PRINT' to display record information is that, if the number of characters exceeds 16, they will be displayed over two lines with the break after the sixteenth character. If the number of characters exceeds 32, then the first 16 characters will be lost as the string scrolls on the display. For small strings or values of less than 16 characters, 'PRINT' would be a satisfactory method.

The 'VIEW' instruction can also be used to display one of the fields. This instruction, you will remember, takes the form 'VIEW(line%,string$)', and like 'GET' used on its own, waits for a (non-cursor) key to be pressed before moving on to the next instruction. With VIEW, *all* of the string is displayed on one line (as dictated by the 'line%' parameter), scrolling if it is over 16 characters. The scrolling can be controlled by using the cursor keys.

To display, on the top line of the screen, the same field-name variable as in the previous example, you would use the instruction VIEW(1,A.ITEM$).

To display a numeric field-name variable using VIEW, you must first convert the numeric variable into a string, of course. (The value of all the conversion instructions in Chapter 3.10 will now be more apparent). Thus, suppose you wished to display the information in the 'PRICE' field of the current record of *logfile* 'A'. The variable – 'A.PRICE' – is a floating point type, and can be converted by 'FIX$' or 'GEN$'. Using 'GEN$', and specifying a length of '12' to cater for the longest possible value, you would write

```
VIEW(1,GEN$(A.PRICE,12))
```

Notice how the conversion instruction GEN$ is embedded in the VIEW instruction: the fact that Organiser allows you to do this saves several lines of programming, and cuts down on the number of variables required. The alternative would be to first assign the conversion to a new string variable (which had been suitably 'declared'), then use *that* string variable in the VIEW instruction.

The VIEW instruction has two advantages over 'PRINT'. First of all, it enables all of the selected field information to be displayed on

one (scrolling) line. Secondly, if any other key is pressed, the ASCII or pattern number for that key can be 'stored' for use – using 'K%=VIEW(1,A.ITEM$)', for example – should such a requirement be needed.

You can also use the 'EDIT' instruction to display a string (or a numeric variable converted to a string). This time, although strings of longer than 16 characters can be displayed on one line, the display does not scroll automatically: you would use the cursor keys to view the 'hidden' characters.

EDIT would be the most suitable instruction to use if you wished to *change* the information being held in a particular record field (see the previous Chapter).

There is another OPL word that allows you to see the *entire* record, with each field occupying its own line. Furthermore, you can use the cursor keys to 'travel' around your record, so that you can examine every part of it – and if any line is longer than 16 characters, it scrolls automatically *when the cursor is on that line.* In other words, the display is exactly the same as the record display used by FIND on the main Menu: all that you cannot do is *edit* any of the information. The word is 'DISP' (short for DISPlay).

DISP must be followed by two parameters, enclosed in brackets and separated by a comma. Thus 'DISP(mode,string$)'. This instruction is like VIEW in that it will return the ASCII or pattern number of any keypress. Thus 'D%=DISP(mode,string$)' is a valid instruction, with the ASCII value of a keypress being stored in 'D%'.

The string$ parameter can be an actual string, between quotation marks, or a string variable – such as 'STRING$' or 'A.ITEM$'.

The 'mode' parameter can be **–1**, **0**, or **1**. Each of these has a different effect, as follows.

DISP(–1,"")

When the mode parameter is '–1', the string parameter is ignored completely – but it must be present (hence the two sets of quotation marks to give a 'null' or empty string). In this instance the *current record* in the *current file* is displayed as detailed in the previous paragraphs: one line to a field; access to any line of the record by means of the **UP** and **DOWN** cursor keys; and when a cursor is positioned on a line which has more than 16 characters, that line will scroll automatically, with control over the scrolling through the **LEFT** and **RIGHT** cursor keys. Pressing any key will cause Organiser to move on to the next instruction. You can arrange your program so that only the **EXE** key causes processing to continue, by a simple routine as follows:

```
WHILE DISP(-1,"")<>13
ENDWH
```

Remember DISP will wait for a key to be pressed, and 'returns' the value of the key. The ASCII' value for **EXE** is '13', so whenever a (non-cursor) key is pressed, 'WHILE' checks to see whether it is equal to 13. If it isn't, then the 'WHILE' instruction is obeyed again (resulting in the record display being 'reset'). When **EXE** is pressed, the condition is no longer true, and Organiser continues with the instruction following 'ENDWH'.

DISP(1,string$)

This operates in a similar way as the previous mode, except that this time, the string$ variable is displayed. On the face of it, there would appear to be little difference between this instruction and VIEW. However, in this instance, if the string includes *tab* characters (ASCII character 9), these break the string up into 'fields', with each 'field' being displayed on its own line. You can create such a string by *concatenating* strings

```
STR1$+CHR$(9)+STR2$+CHR$(9)+STR3$
```

'DISPL(1,string$)' is intended more for *non* file-handling pro-
grams, enabling you to display information on the screen in the same way as a record would be displayed.

DISP(0,"")

This operates the same as the previous two modes – but this time, the **previous** record or string that was displayed using the 'DISP' instruction is displayed again. However, if any other instructions accessing the screen are used between the two uses of 'DISP', the results are somewhat unpredictable. A 'CLS' instruction between the two, for example, could result in some or all of the information being 'missing' until a cursor key is pressed.

Erasing an unwanted record

You will want to be able to erase records from your file from time to time. The OPL instruction to achieve this is simply 'ERASE'.

ERASE is a complete instruction in itself. It removes the *current* record from the *current* file. Once executed, the *next* record in the file becomes the current record – it is just as if a card is removed from a card index box, and all the rest move forward.

If the *last* record in a file is erased, then the EOF flag is set, and the 'current' record will be just a blank.

You will want to include an ERASE routine in any file program

that you write, so that you can remove unwanted records. How you tackle the particular ERASE procedure depends on the nature of the file: one method is to display a relevant field for each record in turn, and then offer the option to erase it or leave it. Such a routine could look like this (it is assumed that the relevant file has been opened as *logfile* 'A.', and that the field to be displayed is 'ITEM$'):

```
FIRST
KSTAT 1
WHILE EOF=0
CLS
AT 1,1
PRINT "ERASE Y/N"
I%=VIEW(2,A.ITEM$)
IF I%=89
  ERASE
  PRINT "RECORD ERASED"
  GET
ELSE NEXT
ENDIF
ENDWH
PRINT "NO MORE RECORDS"
GET
```

This routine uses 'VIEW' to display the field-name string variable, and to get the answer to the question "ERASE Y/N". The ASCII number for capital **Y** is 89 (an alternative way to write the test instruction is 'IF I%=%Y'): the 'KSTAT 1 instruction is used to ensure the keyboard is set for capital letters. Notice that a test is made to see if a **Y** is pressed – but not **N**: any key other than **Y** is taken to mean "no". Notice too that, if the record is not erased, the 'NEXT' instruction must be used – to make the next record the current record. If a record is erased, the next record automatically becomes the current record.

Finally, you'll see that a 'WHILE/ENDWH' loop is used. This is purely to demonstrate how 'EOF' can be used with 'WHILE/ENDWH': as long as EOF is equal to zero, a valid record is current. When EOF is not zero – the flag is set – and the end of the file has been reached. 'DO/UNTIL EOF' can replace the 'WHILE/ENDWH' instructions.

This routine will necessitate stepping through each record in turn, making a decision each time as to whether the record on display should be erased. For long files, this could be tedious. An alternative is to arrange the erase routine so that the records to be deleted are identified by the 'FIND' instruction. Using the same conditions as before, such a routine could look like this:

```
KSTAT 1
MORE::
```

```
FIRST
CLS
PRINT "ERASE CLUE"
INPUT C$
DO
  CLS
  I%=FIND(C$)
  IF I%
    AT 1,1
    PRINT "ERASE Y/N"
    I%=VIEW(2,A.ITEM$)
    IF I%=%Y
      ERASE
    ELSE NEXT
    ENDIF
  ELSE PRINT "END OF FILE"
  GET
UNTIL EOF
I%=MENU("MORE,END")
IF I%=1
  GOTO MORE::
ENDIF
```

Notice how the 'FIRST' instruction is positioned so that each *new* search starts at the first record in the file. Notice too that it is not assumed that the search clue will find the *correct* record to be deleted, nor that *only* one record is to be deleted: the option is given to delete all records matching the search clue. The message "ERASE CLUE" is given rather than "SEARCH CLUE", to remind you that you are performing an ERASE operation. The 'IF I%' instruction enables the erase display to be avoided if I% is equal to zero – a condition occuring when the end of the file is reached. Remember that 'IF' tests to see whether there is a *zero* (for untrue) or a value (for true) in the following 'comparison'. We use I% here *as the comparison*. If I% is zero – representing 'untrue' – a jump is made to the 'ELSE PRINT "END OF FILE"' instruction.

Finally, notice that the alternative method of specifying an ASCII number is used in this routine: '%Y' for the ASCII value of Y. (The range of ASCII numbers is given in the Appendix).

These are just two examples of how you could prepare an *erase* routine for your file. Obviously they can be adapted to suit your own needs – and you will no doubt by now be able to write other routines to suit your own purposes.

Closing a file

When you have finished working on a file, your program should *close* it if you intend to open another with the same *logfile* identification. The OPL word is CLOSE.

CLOSE closes only the *current* file: amongst other things, that means the file-name variables in that file are no longer available to you.

If more than one file is opened, *each one* must be closed by a separate 'CLOSE' instruction if it is intended to open new files.

If a program ends without files being closed, they are closed automatically.

Remember...

You can work only on files that have been *opened*.
Only *one* file is *current* for working on (for 'FIND', 'ERASE' and similar operations).

The field-name variables for all *opened* files are accessible all the time – but the values in those variables will relate to the *current* record number or position, unless you have assigned them otherwise.

3.16

FILE MANAGEMENT

OPL words covered
COPY, DELETE, DIR$, RENAME, SPACE

Looking after your files

From time to time, you may wish to copy a file from one location in memory to another – from RAM to a Datapak, for example. You may also wish to examine the names of all the files you have created, and possibly rename some or delete those you have no further use for. It will also be useful to know how much space is left on a Datapak (you can find out how much space is left in RAM by entering **FREE** in the CALCulator mode).

All of these operations come under a general heading of 'File management' – and whilst it is possible to achieve any of them by including routines in your file handling programs, it can be easier if they are combined in one management program. We shall develop such a program throughout this Chapter.

Examine file names

You will be aware that the PROGramming Menu of Organiser has a DIRectory option, enabling you to step through the names of all your procedures saved in RAM or on a Datapak.

There is an OPL word 'DIR$', which will allow you to examine the *files* you have saved. The complete instruction takes one of two forms:

```
DIR$(location$)
DIR$("")
```

Using 'DIR$(location$)' displays the name of the *first* file to be found
at the specified location – "A" for RAM, "B" or "C" for the upper and
lower Datapaks respectively. The 'DIR$("")' instruction displays the
next file to be found at the previously specified location. Thus, both instructions need to be used in order to display all the files at a specified location – 'DIR$(location$)' first, then 'DIR$("")' repeatedly.
When there are no more files to display, 'DIR$("")' returns a null or

'empty' string.

Here is a procedure using the DIR$ instruction, which will enable you to examine the files you have saved in RAM or on a Datapak.

```
FDIR:
LOCAL G%,L%,A$(2),B$(10)
ONERR MORE::              :REM See next Chapter
L%=0
LOC::
L%=L%+1                   :REM For location letter
IF L%>3
  L%=1                    :REM Only three locations
ENDIF
MORE::
A$=CHR$(64+L%)+":"        :REM = "A:","B:" or "C:"
CLS
CURSOR ON
PRINT "DIR",A$;           :REM Display DIR and letter
G%=GET
IF G%=2                   :REM Means MODE key pressed
  GOTO LOC::              :REM so change letter
ELSEIF G%=1               :REM Means CLEAR/ON pressed
  GOTO RET::
ELSEIF G%=13              :REM Means EXE pressed
  CLS
  DO
    B$=DIR$(A$)           :REM First time = letter
    PRINT B$              :REM Display 1st file name
    IF B$=""              :REM No more files
      PRINT "NO MORE FILES"
      GET
      GOTO RET::
    ENDIF
    A$=""                 :REM For next file name
    G%=GET                :REM Display until keypress
  UNTIL G%=1 OR B$=""     :REM 1 means CLEAR/ON=stop
ELSE GOTO MORE::
ENDIF
RET::
CURSOR OFF
ONERR OFF
```

Program 3.16.1 'FDIR' File name Directory

This procedure will be used in our File Handling program. As before, you do not need to enter the leading *indent* spaces, nor the ':REM' statements - they should be sufficient for you to understand how the procedure operates.

When it is run, pressing the **MODE** key will enable you to select

one of the three possible locations (RAM or Datapaks) – provided of course you have Datapaks fitted. If you do not have a Datapak fitted in the chosen location, an error will occur which would stop the procedure running and break you out of the program. This has been avoided by using an 'ONERR' instruction at the beginning, and ensuring that all 'exits' from the procedure are via an 'ONERR OFF' instruction. These two new OPL words are discussed in the next Chapter.

Once the appropriate location has been selected, pressing **EXE** will step through all the files saved to that location, until the last one, or until **CLEAR/ON** is pressed to leave the procedure.

Datapak space free

If you have Datapaks fitted, you will no doubt wish to know at some point how much of the memory is still available for you to use. The OPL word to help you do this is is 'SPACE'.

SPACE returns the number of free bytes on the Datapak with the *current* opened file. This means that if you wish to find out how much free space there is on a Datapak, you must first open a file and make it the current file. Even if you have not created any files on a Datapak – you have only program procedures on it, for example – this is still possible, because Organiser automatically creates a file on every new Datapak fitted. This is the 'MAIN' file – for use from the main Menu.

As with the previous procedure, it is necessary to ensure that errors do not cause Organiser to break out of the procedure.
Here is a procedure to tackle the task:

```
FSPACE:
LOCAL L%,G%,A$(2)
ONERR MORE::
L%=0
LOC::
L%=L%+1
IF L%>3
  L%=1
ENDIF
MORE::
A$=CHR$(64+L%)
CLS
CURSOR ON
PRINT "LOCATION",A$
G%=GET
IF G%=2                 :REM MODE key
  GOTO LOC::
ELSEIF G%=1             :REM CLEAR/ON key
```



```
  ONERR OFF
  CURSOR OFF
  RETURN
ELSEIF G%=13            :REM EXE key
  CLS
  OPEN A$+":MAIN",A,T$    :REM MAIN at location
  PRINT SPACE           :REM Print free space
    GET
    CLOSE               :REM Close in case more
ENDIF
GOTO MORE::
```

Program 3.16.2 'FSPACE' Memory left on a Datapak

This procedure follows the same lines as the previous procedure, and shouldn't need any further explanation. When run, you will be able to pick the location using the **MODE** key, then find out how much free memory there is by pressing the **EXE** key.

Note that you can also use the procedure to give the free space in RAM – but, do bear in mind that this procedure *at least* will be loaded into the running area, so reducing the free-space available. You can check more accurately on how much space there is in RAM using FREE in the CALCulator mode: the difference between the value given by FREE and the value given by this procedure will show you how much space this procedure and any calling it take up in RAM (in the *translated* version).

Deleting a file

The OPL word to perform this operation is 'DELETE', and it must be followed by the complete file name – including the location – within quotation marks, or as a variable. Thus to delete a file called 'TEST' saved on the upper slot Datapak, you would write

```
DELETE "B:TEST"
```

Here is a procedure to perform the operation. This time, the location and file-name must be entered at the keyboard, and so we will include a check that it has been entered correctly.

```
FDEL:
LOCAL G%,F$(10)
NAME::
CLS
PRINT "FILE TO DELETE";CHR$(63)  :REM=Question mark
INPUT F$
CLS
T$=UPPER$(LEFT$(F$,1))
IF F$=""                    :REM Escape route
```

```
  RETURN
ELSEIF (T$<"A") OR (T$>"C") OR (MID$(F$,2,1)<>":")
  PRINT "INVALID LOCATION"
  GET
  GOTO NAME::
ELSEIF EXIST(F$)            :REM Is there a file
  DELETE F$                 :REM Yes, so delete it
  PRINT F$
  PRINT "DELETED"           :REM and say so
  GET
ELSE PRINT "FILE NOT FOUND"
GET
GOTO NAME::
ENDIF
```

Program 3.16.3 'FDEL' Delete a file

Notice the 'escape' route, should the procedure be selected inadvertently: simply pressing **EXE** when the file name is requested will exit the procedure. Notice too how a check is made that a valid location identifier precedes the actual file name.

This procedure allows you to delete only one file at a time: if you wish it to handle more, you can put a 'GOTO NAME::' instruction following the 'GET' (after 'PRINT "DELETED"') instruction.

Warning – don't delete the 'MAIN' file – else Organiser will have nowhere to save your main Menu records! (And you won't be able to use the 'FSPACE' routine as it is written).

Be careful what files you delete: once deleted, they are gone forever!

Copying files

The OPL word is 'COPY', and it operates in much the same way as the COPY on the main Menu. The COPY instruction takes three forms:

COPY "loc1:filename1","loc2:filename2"
This copies 'filename1' at location 'loc1', to a new file 'filename2' at location 'loc2'

COPY "loc1:filename1","loc2:"
This copies 'filename1' at location 'loc1', to a file of the same name at location 'loc2'.

COPY "loc1","loc2"
This copies *all* the files at location 'loc1' to the location 'loc2', retaining the same file names.

The important points to note are:

a) The *copied* file remains at location 'loc1': if you want it deleted, you must do so separately.
b) You cannot copy files back to the same location – even with a changed file name.
c) If the named file already exists at the destination location 'loc2', then the records from the file at 'loc1' are *appended* to it. Otherwise a new file is created, and the records are written into it. This can therefore be a useful way of adding new records created in RAM to an existing file on a Datapak.

Just as with the other file management facilities discussed in this Chapter, it is possible to write a procedure for copying files using the OPL word 'COPY'. However, the procedure would merely duplicate the COPY option on the main Menu - and take up valuable space in your Organiser. It is suggested, therefore, that to copy a file or files from one location to another, you use the main Menu facility.

The copying process can take as long as several minutes, depending on the amount of information being copied, and places an extra drain on the battery. ***Make sure your battery is reasonably fresh before attempting the process, or alternatively, use the special mains adaptor.*** If the power fails during a COPY operation, any record partially copied will be deleted. The others will remain intact.

Note that COPY, as provided by the main Menu or by the OPL instruction, is for *files*, whereas the COPY option on the PROGramming Menu is for *procedures*. Don't confuse the two!

Renaming files

There may be occasions when you will want to rename one of your files. The OPL word is 'RENAME', and the full instruction has the following format:

```
RENAME "loc:oldname","newname"
```

Thus, to rename a file called 'TEST' in the upper Datapak as 'GOOD', you would write 'RENAME "B:TEST","GOOD"'. If the location of the file is omitted, Organiser will assume the file is in *current* location – or if no files are being dealt with, in RAM (location 'A:'). If you have files of the same name in different locations, this could result in the wrong file being renamed. Hence it is advisable to ensure that the location is properly entered along with the file name to be changed. Here is a procedure to perform the task:

```
FREN:
LOCAL F$(10),N$(8)
START::
CLS
PRINT "NAME TO CHANGE"
INPUT F$
N$=UPPER$(LEFT$(F$,1))
IF F$=""
  RETURN                  :REM Escape route
ELSEIF (N$<"A") OR (N$>"C") OR (MID$(F$,2,1)<>":")
  PRINT "INVALID LOCATION"
  GET
  GOTO START::
ELSEIF EXIST(F$)          :REM Must exist to rename
  NEW::
  CLS
  PRINT "NEW NAME"
  INPUT N$
  TRAP RENAME F$,N$       :REM New words are used here
  IF ERR                  :REM to act on errors
    PRINT ERR$(ERR)       :REM - see next Chapter
    GET
    GOTO NEW::
  ENDIF
ELSE PRINT "FILE NOT FOUND"
  GET
ENDIF
```

Program 3.16.4 'FREN' Rename a file

This procedure uses some new words - 'TRAP', 'ERR' and 'ERR$'. These are discussed in the next Chapter, but briefly, 'TRAP' prevents any error resulting from the 'RENAME' instruction causing Organiser to stop the program to report the error, while 'ERR' and 'ERR$' are used to detect an error and print the error message. These words save you from trying to write routines to test for every possible mistake that can be made when entering the actual file names.

Notice the 'escape route' in the event of the procedure being selected inadvertently: pressing **EXE** will exit the procedure.

The main File Management procedure

We will now prepare a short procedure to run the three previous file management procedures. You will have noticed that there are elements in each procedure that are common: the overall program could be shortened by making these common elements separate procedures, defining the variables they use as **GLOBALS** in the main controlling procedure. (This is left for you to do, as an

exercise in programming, should you so wish). The individual procedures have been made 'self-contained' deliberately, for those who wish to add only one or two of them to their program stock.

```
FMAN:
LOCAL C%
OPT::
C%=MENU("FILES,SPACELEFT,RENAME,DELETE,END")
IF C$=1
  FDIR:
ELSEIF C%=2
  FSPACE:
ELSEIF C%=3
  FREN:
ELSEIF C%=4
  FDEL:
ELSE RETURN
ENDIF
GOTO OPT::
```

Program 3.16.5 'FMAN' File management main procedure

Obviously all the called procedures must have been entered and saved for this procedure to operate. Copied to a Datapak as 'OBJECT ONLY', the entire program will occupy approximately 1kbyte.

3.17

ERRORS AND BUGS

OPL words covered
ERR, ERR$, ONERR/OFF, RAISE, STOP, TRAP

When your slip shows

Unfortunately, there are a number of things that can go wrong when writing and running programs. In very broad terms, these can be grouped into two categories: incorrectly used instructions, and incorrect program *logic*.

As far as incorrectly used instructions are concerned, some will be discovered when a procedure is translated (a missing 'UNTIL' or 'ENDIF', for example), while the others – such as variables that have not been declared or are incorrectly assigned – will be discovered when the program is run. Organiser has a comprehensive error-reporting system built in to identify very closely the type of error that has occured in these circumstances. Generally speaking, these errors must be corrected by editing the appropriate procedure, so that the program can run without stopping.

What Organiser *cannot* do is assess what was in your mind when you planned and wrote the program: it follows your instructions implicitly as far as it can. So if, for example, your program allows Organiser to run in a perpetual 'loop', then in a loop it will run. The problem is that it is not always easy to identify a situation that *causes* the program to get into a loop: the larger the program, the more difficult it is to test for *every* possible eventuality that could cause trouble.

If the program requires user inputs – and most programs do – an incorrect input or an input that hasn't been catered for can cause the program to 'go wrong' with unpredictable results.

There is a considerable overlap between errors of *logic* and errors involving incorrect use of instructions. For example, you may have a program which divides one value by another. If by some unforeseen event the second number is a *zero*, Organiser will stop to report a 'DIVIDE BY ZERO' error. On the face of it, Organiser has been given an instruction it cannot perform, and so it stops to report it. In fact, the *logic* of the program could be wrong, in that it permits a zero value for the second value under certain conditions. Your problem then is to identify *why* a zero occured in the second value.

The OPL set of instructions includes words that allow you to prevent any or all of the errors Organiser is capable of reporting from stopping a program running. This feature enables all the most

common troubles – such as dividing by zero – to be dealt with so that the program *doesn't* stop running: instead, *you* are put in control of any messages that are be displayed, and the course of events that follow. This does mean, of course, that you take over the responsibility for dealing with the errors concerned with your own instructions, and so the facility must be used with great care.

We saw a crude example of *error trapping* (as the process is called) in the 'SPACE' procedure given in the previous Chapter. When running this procedure, it is quite possible to attempt to find out how much free space there is available on a non-existent Datapak. Normally, Organiser would break out of the program to report such an error. However, the program allows for such an eventuality (using the 'ONERR' instruction) so that, should it arise, a jump is made back to the 'location selection' routine. No message is given: you simply are not allowed to select an 'incorrect' location. The error is 'trapped' and dealt with in the program.

Where an error in the *logic* of a program produces an error that Organiser can detect, it can be trapped. Unfortunately, there is no short-cut to locating the type of error that cannot be trapped. There are, however, various techniques that can be employed to help make the process of finding such an error a little less frustrating. These will be discussed later: first, let us look at the errors that Organiser is capable of reporting.

The error messages

Organiser has a very extensive range of error messages. Each message has two components: a *number*, which identifies the error type, and the message itself.

As mentioned earlier, Organiser can detect some types of error when it is *translating* your procedure into the language it uses to run the program. These errors will be related to misused or missing instructions, resulting in Organiser being unable to make the translation. Here is a list of such errors in alphabetical order, together with the 'error number', a brief description and an indication of the course of action to take:

Errors occuring during the translation process

215 **BAD ARRAY SIZE.** The number of elements declared in an array is incorrect: the number must be higher than zero.

208 **BAD ASSIGNMENT.** An attempt has been made to assign a value to parameter defined as an input to a procedure: these parameters can be used in the procedure, but not changed. The procedure should be corrected.

216 **BAD DECLARATION.** A variable has not been declared properly: perhaps the space to be allocated to a string variable has not been specified, or a numeric array has been given two or more parameters.

207 **BAD FIELD LIST.** The number of fields specified for a file is outside the range 1 to 16.

222 **BAD IDENTIFIER.** An identifier is used incorrectly – 'TEST%%', for example.

209 **BAD LOGICAL NAME.** The *logfile* letter is not A, B, C or D.

197 **BAD PROC NAME.** This error occurs when a new procedure is being named, if it doesn't follow the rules.

214 **DUPLICATE NAME.** A variable has been declared twice. This error can also occur if a file or a procedure name is duplicated at the specified location.

227 **MISMATCHED ()'s.** You have used too few or too many brackets in an expression: check that the number of *left* brackets is the same as the number of *right* brackets – and that they are in the correct places.

221 **MISMATCHED "**. You have used too few or too many quotation marks round a *string*.

210 **MISSING COMMA.** A comma is missing from a list of items – usually a list of parameters which should be separated by commas.

211 **MISSING LABEL.** A 'GOTO' or an 'ONERR' instruction is followed by an incorrectly defined label (it must end with two colons), or the label does not exist in the procedure.

223 **NAME TOO LONG.** A variable has been given a name that exceeds eight characters including the identifying symbol. Note that this error can also occur if a file name or a procedure name is too long.

254 **OUT OF MEMORY.** You have run out of memory space – see the *Errors while running* section that follows.

213 **STRUCTURE ERROR.** A word in the groups IF/ENDIF, WHILE/ENDWH, or DO/UNTIL has been missed out, or an extra one inserted.

228 **SYNTAX ERR.** An error in the *syntax* of an instruction – the spelling of an OPL word is incorrect, perhaps, or the punctuation is not what Organiser expects.

212 **TOO COMPLEX.** There are too many 'nested' structures (IF, DO, WHILE): the limit is eight 'depths'.

224 **TYPE MISMATCH.** A string has been assigned to a numeric variable or vice versa. This error can also occur when a procedure parameter is given an incorrect parameter type – which may be detected only when the procedure is run.

One or two of these error may also occur when a program is running. When they occur during the TRANslation process, Organiser will stop translating your procedure. You must correct the error before the procedure can be translated properly. There is little point in trying to 'trap' this type of error: the procedure can never get to the 'running' stage if it exists.

Here are the errors that can occur while a program is running. Again, they are presented in alphabetical order, with the error number and a brief description.

Errors occuring when a program is running

205 **ARG COUNT ERR.** An incorrect number of parameters ('arguments') has been passed to a procedure. The procedure should be edited.

219 **BAD CHARACTER.** A non valid character has been used in a calculation or an 'expression'. The procedure should be corrected.

231 **BAD DEVICE CALL.** This error occurs when a machine code program is run, indicating that the specified location is not available.

243 **BAD DEVICE NAME.** A location other than A, B or C has been specified.

236 **BAD FILE NAME.** The name given to a file does not obey the rules – see Chapter 3.13.

226 **BAD FN ARGS.** An incorrect number or wrong types of parameter (argument) have been used when calling a function. For example 'LOG(2,3)' instead of 'LOG(2.3)'.

218 **BAD NUMBER.** Organiser has met a number that it cannot evaluate properly – perhaps because it contains too many digits.

197 **BAD PROC NAME.** When creating a new procedure, it has been given an invalid name. See Chapter 3.2 for the 'rules'.

237 **BAD RECORD TYPE.** This error message should occur only when running machine code programs – which are beyond the scope of this book.

229 **DEVICE LOAD ERR.** A program pack or a peripheral device has been removed whilst it is being loaded or used, or the information has been 'corrupted' in some way.

230 **DEVICE MISSING.** An attempt has been made to access a device which is no longer connected to Organiser – a printer or a bar-code reader, for example.

233 **DIRECTORY FULL.** The maximum number of files and procedures that can be saved at any location is 110, even though there may be memory space for more. This error message tells you that number has been reached: either copy some to another location, or erase some to make *directory* space available.

251 **DIVIDE BY ZERO.** As it says, an attempt has been made to divide a number by zero.

238 **END OF FILE.** Occurs during a file handling operation when the end of the file is reached: use of the 'EOF' flag can avoid this error occuring – see Chapter 3.15.

206 **ESCAPE.** Not, technically, an error since it appears when **CLEAR/ON** followed by **Q** has been pressed, to deliberately stop the program from running. Pressing **CLEAR/ON** merely stops the the program temporarily: it can be re-started by pressing any key other than **Q**. Note that if the 'ESCAPE OFF' instruction has been used, it will not be possible to stop the program using **CLEAR/ON** and **Q**.

253 **EXPONENT RANGE.** In scientific notation, numbers must lie within the exponent limits of –99 to +99. The program has a number outside of these limits.

201 **FIELD MISMATCH.** A 'field name' variable has been used which does not match any of the field names in any of the *opened* files. The procedure must be edited to correct the error before it can run properly.

235 **FILE EXISTS.** An attempt has been made to create a file or procedure with the same name as one already saved at the specified location. Think of a new name – or, for file handling routines, make use of the 'EXIST' instruction (Chapter 3.13).

199 **FILE IN USE.** The procedure is trying to open a file which has already been opened, or to delete a file which has been opened.

234 **FILE NOT FOUND.** An attempt has been made to EDIT a procedure or to open a file which does not exist at the specified location. For file handling routines, the 'EXIST' instruction can be used to avoid this error stopping program running (Chapter 3.13), or error-trapping can be used.

196 **FILE NOT OPEN.** Your procedure is trying to work on a file that has not been opened. The procedure will need editing to correct the error.

247 **FN ARGUMENT ERR.** The parameter information (the 'argument') passed to a function or a procedure is of the wrong type – an integer instead of a floating point, perhaps. Or it could be a negative value where only positive values can be accepted: L = LOG(–1), for example.

195 **INTEGER OVERFLOW.** Integers can have values between –32768 and +32767 only: this range has been exceeded in the procedure.

202 **MENU TOO BIG.** The string yielding the Menu display in the MENU instruction is too long for Organiser. The procedure must be edited to correct the error – by shortening the string – before it can run properly.

204 **MISSING EXTERNAL.** Organiser has come up against a variable that it cannot find anywhere among the 'LOCAL' or 'GLOBAL' declarations for the procedure, or among the 'GLOBAL' declarations for a *calling* procedure. With this message, Organiser will tell you the name of the variable it cannot find and, when **SPACE** is pressed, the procedure where it is being used. Pressing **SPACE** again will give you the option of editing the procedure (as long as the procedure has not been saved as *object only*) so that you can declare the variable within that procedure – or perhaps correct its name. It may be that you wish to declare the variable as a 'GLOBAL' in a different procedure that

calls the one in which the error occured. Whatever the reason for the error, the program must be corrected before it can run properly.

203 **MISSING PROC.** A procedure has been called that Organiser cannot find anywhere in RAM or on a Datapak: has the Datapak been removed? Is the procedure named correctly? Have you written it yet? Whatever the cause, it must be put right before the program can run properly.

255 **NO ALLOC CELLS.** This will occur only with *machine code* routines accessing the Organiser's internal buffer space. Machine code routines are beyond the scope of this book.

246 **NO PACK.** An attempt has been made to use a Datapak when there is no Datapak fitted. You can't pull the wool over theOrganiser's eyes.

217 **NO PROC NAME.** A program developed outside of Organiser has an invalid procedure name as its first line.

250 **NUM TO STR ERR.** This error occurs when Organiser's operating system routines are 'called' from a program, which is beyond the scope of this book. It could also be seen when making CALCulations.

254 **OUT OF MEMORY.** All the available space in Organiser's RAM has been used up, and there is no more room to do anything else. You must create some space if you wish to continue running the program – by deleting files or file records, tidying up the Diary, and so on.

242 **PACK CHANGED.** A Datapak has been changed in the middle of copying information onto it.

239 **PACK FULL.** The Datapak selected for saving information has no memory left: all of the valid information it contains should be copied to a fresh Datapak, and then it should be re-formatted. (Contact your dealer or Psion).

241 **PACK NOT BLANK.** The Datapak has residual information on it, and it needs to be re-formatted – i.e. cleared of all its information.

232 **PAK NOT COPYABLE.** Program packs are often protected so that you cannot make 'illegal' copies. An attempt has been made to copy such a pack. Naughty you.

244 **READ ONLY PACK.** An attempt has been made to write information to a Datapak that will not accept information in Organiser II: it may be a protected *program* pack.

200 **READ PACK ERROR.** Organiser is unable to read the information contained on a Datapak: the Datapak will have to be re-formatted.

198 **RECORD TOO BIG.** An attempt has been made to put more than 254 characters into a record – 254 is the maximum size, however many fields (up to 16) there may be. Build a test or error trapping routine into the procedure, to ensure the limit is not exceeded.

249 **STACK OVERFLOW.** This error occurs when a machine code routine destroys a specific area reserved for Organiser's use: such routines are beyond the scope of this book.

248 **STACK UNDERFLOW.** See above.

220 **STRING TOO LONG.** A string of characters is too long for the string variable to which it is being assigned. The maximum number of characters a string variable can contain is specified when it is declared with the 'LOCAL' and 'GLOBAL' instructions.

252 **STR TO NUM ERR.** An attempt has been made to convert a string that contains characters other than numbers to a number. For example `VALUE%=VAL("123A4")`.

225 **SUBSCRIPT ERROR.** A non-existent array element has been specified: arrays can never have a zero element (such as 'ARRAY(0)'), and can never have elements greater than the declared number.

240 **UNKNOWN PACK.** A Datapak that Organiser II doesn't recognise has been plugged in. Only Datapaks formatted for Organiser II should be used. (Note that Organiser I Datapaks can be *read*, but not *written to*.

245 **WRITE PACK ERR.** For some reason, Organiser cannot write information to a Datapak. Perhaps it isn't fitted properly or, if you're using a mains adaptor, perhaps there's been a hiccup in the mains supply.

Should any of these errors occur when running a program, the relevant procedure must be corrected. In some instances, the correction may take the form of a *trap* followed by a specific course of action, to enable the error to be corrected while the program is running.

Organiser II provides two different methods for *trapping* errors. One is a 'global' trap – which operates on *any* error that may occur, while the other operates to trap errors that can occur when specific OPL words are used.

Trapping any error

The OPL instruction to divert processing to an error handling routine should any of the listed errors occur is `ONERR`, which must be followed by a 'label': thus `ONERR WRONG::` would be a typical instruction. The label *must* be found elsewhere in the same procedure.

When the `ONERR` instruction is used in a procedure, any error occuring *after* that point – even in subsequently called procedures – will cause the processing to jump to the specified label in the original procedure. Should you wish errors in called procedures to be handled differently, then the `ONERR` instruction must be used again with a different label. You can switch off the `ONERR` facility, by the instruction `ONERR OFF`.

Whilst this facility will trap any of the OPL listed errors that can occur, you will no doubt wish to use it to trap specific types of error. The 'SPACE' procedure (3.16.2) is an example of this type of use. The only likely error to occur in this procedure (once it has been thoroughly tested) is the selection of a Datapak location when a Datapak is not connected. The processing is re-directed on such an error to the point where the location is selected. You may, however, wish to include an error message.

To give an example, here is a short procedure that can be run from the main Menu, to find the logarithm of a number.

```
LOG:
LOCAL L
PRINT "NUMBER="
INPUT L
PRINT LOG(L)
GET
```

This is a demonstration procedure which will return to the Menu after finding the logarithm of one number. If it were installed on the main Menu and an error occured (an incorrect value for 'LOG' is entered), the procedure would stop, Organiser would display the error message, and you would be returned to the main Menu when a key is pressed. You would then have to run the procedure again.

With a longer program, this could be inconvenient. By using the 'ONERR' facility, however, the error can be 'trapped':

```
LOG:
LOCAL L
GOTO START::
WRONG::                    :REM Error handling part
ONERR OFF
CLS
PRINT "INVALID ENTRY"
GET
IF GET=1                   :REM CLEAR/ON key
  RETURN                   :REM Escape route
ENDIF
START::
ONERR WRONG::
PRINT "NUMBER="
INPUT L
PRINT LOG(L)
GET
```

Program 3.17.1 'LOG' Demonstration of ONERR

In this procedure, a 'jump' is made over the error-handling routine 'WRONG::'. Should an error occur because a value of zero, a negative value or a character is entered by the user (for which there is no 'LOG') – processing will jump to 'WRONG::', report that an invalid entry has been made, then follow through for a further input – unless the **CLEAR/ON** key is pressed to abandon things. Note that error handling is switched off ('ONERR OFF') at the beginning of the error-handling routine: should an error occur in this part, perhaps because Organiser runs out of memory space when the PRINT instruction is encountered, the procedure will run into a loop. The escape route – if **CLEAR/ON** is pressed after the "INVALID ENTRY" message is displayed – ensures you can break out of the procedure come what may.

The error handling routine *could* have been added at the end of the procedure. In this case, it would be absolutely vital to include a 'RETURN' or some other way of escaping the procedure after the answer has been displayed. Otherwise, having performed the operation, processing would continue with the error handling routine – with a jump back to 'START::' – and Organiser would be in
a perpetual loop from which there is no escape.

It is important to note that the* CLEAR/ON *and* Q *method of breaking a program will not work whilst Organiser is waiting for an 'INPUT': the* CLEAR/ON *key, in these circumstances, serves to clear any entered input.

The actual type of error can be identified and displayed on the screen – just as if Organiser were reporting the error – by using the two OPL words 'ERR' and 'ERR$'.

When an Organiser-detectable error occurs, the error *number* is 'stored' in ERR. Thus, a specific error can be isolated by using the 'IF' construction: 'IF ERR=251', for example, would isolate the 'DIVIDE BY ZERO' error.

ERR$ must be followed by an error number, in brackets, and gives the message string for the specified number. Thus, an instruction such as 'PRINT ERR$(251)' will display on the screen the message 'DIVIDE BY ZERO'. Using the two words 'ERR' and 'ERR$' in combination – 'PRINT ERR$(ERR)' – in an error handling routine allows the Organiser error message to be displayed, whatever it may be.

If a number *outside* Organiser's range of error numbers is used, the screen will simply display the message ***** ERROR *****.

When writing and testing your procedures and programs, you can therefore cater for various types of error that may occur when the program is run. Obviously you would not cater for any *file-handling* errors in a program that does not use files. Having written your procedure to cater for errors – either at the outset, or by editing it as a result of the error occuring during the program testing – you may wish to test that the program operates correctly when an error *does* occur.

There is an OPL word to allow you to do this – RAISE. This must be followed by an error number: 'RAISE 251', for example, will artificially generate a 'DIVIDE BY ZERO' error.

The 'RAISE' instruction is added at a suitable point in the procedure – *outside* of the error handling routine – when you are testing that it works. If there is no error handling routine, then processing will stop, just as if the error had actually occured, with the appropriate message displayed on the screen.

Remember that 'ONERR' operates on any procedures called by the procedure in which it is used, and any procedures *they* may call, and so on. Processing will always return to the 'top' procedure, containing the 'ONERR' label. (This could be used as a very fast way of jumping back from a 'low level' procedure to the top procedure).

If you want to isolate different types of error in each individual procedure, then each must have its own routine. If you don't want error handling to continue through to called routines, then you must 'switch off' the trapping process, using 'ONERR OFF'.

Trapping specific instruction errors

There are many types of error that can occur during file handling – because a specified file does not exist or has not been opened;

because an incorrect location or *logfile* has been specified; because the end of the file has been reached; because an incorrect user input has been made, and so on.

Organiser allows you to trap errors of this kind with the OPL instruction 'TRAP'.

TRAP can be used in front of *any* of the following instructions:

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

Of these, all except 'INPUT' and 'EDIT' deal only with file handling.

Typical instructions might be 'TRAP APPEND', or 'TRAP NEXT'. Using TRAP prevents Organiser from stopping the program running if an error results from associated instruction. Thus, with 'TRAP APPEND', if an error resulted from the 'APPEND' instruction (perhaps because a file is not open) then Organiser would not stop to report the error.

It is up to you to tell Organiser what it must do if an error occurs.

Generally speaking, you will know what errors can result from an instruction – either by experience, or as a result of program testing. Immediately after the 'TRAP xxxx' instruction, you must put in your error handling routine, which will invariably start with an 'IF ERR...'. Then, if an error does occur, it is dealt with immediately. If there is no error, the 'IF ERR...' is ignored and processing will continue with the instruction following the routine terminator 'ENDIF'.

Here is an example, using TRAP to prevent an error occuring during an 'INPUT' instruction. It would be well worth your while to enter this short procedure, test it out and examine its operation, so that you can understand the process involved.

```
TRAPPER
LOCAL I%
START::
CLS
PRINT "ENTER A NUMBER"
TRAP INPUT I%
IF ERR
  CLS
  PRINT "ERROR No.",ERR
  PRINT ERR$(ERR),
  GET
  GOTO START::
ENDIF
```



```
PRINT "THAT IS A NUMBER"
GET
```

Program 3.17.2 'TRAPPER' Demonstration of TRAP instruction

When you run this procedure, the keyboard will be set for numeric inputs on the 'TRAP INPUT' instruction – because of the 'I%' variable. If you press **SHIFT** and a number key, or you press any of the *non* numeric keys (< or =, for example), or the **CLEAR/ON** key, then there will be an error. Because the 'TRAP' instruction has been used, Organiser will not stop to report the error – but carries on with the next instruction. Here, a general 'IF ERR' instruction has been used – rather than isolate a specific error type. The error number is then displayed on the screen along with Organiser's own error message. A jump is then made back to get a proper numeric input.

The only way out of this procedure is to enter a proper number: in longer error handling routines, or routines which pick up an unexpected error, you should provide an alternative 'escape' route, otherwise you will be in a perpetual loop. This is especially important if your procedure doesn't cater for the error that has occured. With the above procedure, you will get error number '252' – 'STR TO NUM ERR' – if you press any non-numeric key, because you are attempting to assign a character to an integer variable. If you enter a floating point number there will be no error, but I% will hold only the *integer* part of the number you enter. If you press **CLEAR/ON** instead of entering a number, a *different* error will be reported – number 206 'ESCAPE'. Without the 'TRAP' instruction, pressing **CLEAR/ON** would not be regarded as an error – but merely an instruction to clear the entry made so far. The fact that **CLEAR/ON** produces an error when 'TRAP' is used with 'INPUT' can be a way of escaping from an error routine. The error handling routine could look like this, for example:

```
IF ERR=206
  PRINT "LEAVING PROGRAM"
  GET
  RETURN
ELSEIF ERR
and so on
```

This will display the message 'LEAVING PROGRAM' if **CLEAR/ON** is pressed when an input is expected. Then, on pressing any key, the procedure will be terminated by the 'RETURN' instruction.

If the procedure is called by another procedure, a return will be made to the calling procedure. There is another OPL word that allows you to break out of the program altogether – and that is 'STOP'. You can use STOP, on its own, wherever you would otherwise use 'RETURN'.

The way you handle errors and the actions that are taken will

depend on your program: remember that when you use error-trapping the onus is on you to tell Organiser what to do should the errors occur. Cater for all potential errors, and always provide an escape route to avoid continuous loops, *especially* during program development. To see Organiser 'seize up' when a considerable amount of information and a number of procedures have been entered can be a wigmaker's delight – to put it mildly.

When Organiser goes into a never-ending loop

It is a programmer's nightmare when the computer gets locked into a permanent loop, repeating an instruction sequence ad infinitum. In many instances, you will be able to break out of the program by pressing **CLEAR/ON** followed by **Q**. However, if it is a short loop including an 'INPUT' instruction, this will not work unless your program allows for it.

Sometimes (*but don't bank on it*), you can save the situation by removing the battery for only about 20 seconds: with luck, when you switch on again, you will be presented with the main Menu, and all of the information and procedures you have saved in RAM will still be intact.

If these two measures fail, then all you can do is remove the power supply long enough to clear RAM completely, and start again.

The best cure for a permanent loop is prevention. Always endeavour to ensure that there is a way out of loop structures, particularly those using the 'INPUT' instruction.

Finding bugs

Anything that goes wrong with a program is called a bug. If Organiser reports what has gone wrong, then the type of bug that you are looking for is more easily located, by examination of the instructions.

The bugs that are not so easy to find are those where the program produces wrong results – an answer is given which you know is incorrect.

It is often necessary to add routines to the program in order to help locate where a bug is occuring and to identify why it is occurring. At strategic points, for example, you could include 'PRINT' instructions to display the values or contents of variables that you are using. Examining the contents of variables in this way lets you see what is actually happening as Organiser obeys your instruction set. The routine that causes the trouble can then be isolated and examined carefully.

The 'IF' instruction can also be used in conjunction with 'PRINT' to help isolate a bug, by making the program report when variables become erroneous. 'EDIT' can be used to change the contents of a string variable – and so on. All the OPL words are at your command

to help you isolate running problems. Remember to remove the added routines once the bug has been cleared.

Where error-handling routines are concerned, the 'RAISE' instruction can be used to generate errors, and so check the operation of the routines.

De-bugging a program is rather like detective work: the facts are all there. They need to be winkled out, carefully and methodically. When found, of course, the program needs to be edited to correct the bug. It helps to have good documentation for your procedures and programs – flowcharts and listings. Of course, as far as listings are concerned it helps to have a printer and an RS232 link.

As mentioned before, with very long and involved programs it is almost impossible to cater for every eventuallity at the outset: some bugs may appear only under very specific running conditions, which have not been allowed for.

It is for this reason that a program should not be saved to a Datapak as *object only* until it is known that *all* potential bugs have been taken into consideration: programs saved as 'object only' cannot be edited. They have to be re-written. And that can be a very tedious exercise.

3.18

MACHINE LEVEL INSTRUCTIONS

OPL words covered
ADDR, ESCAPE OFF/ON, PEEKB, PEEKW
POKEB, POKEW, USR, USR$

A word of caution

The OPL words discussed in this Chapter enable the user to gain 'access' to Organiser's operating system. Consequently, they should be used with caution: careless use can result in the loss or corruption of saved information, and in the operating system behaving abnormally. They cannot damage the *hardware* of Organiser, and consequently any effects resulting from misuse can be cleared by removing the power supply and reconnecting it after a suitable period of time. The golden rule is – if you don't understand what the instructions are doing, don't use them.

With one exception, an appreciation of *machine code* language is needed in order to fully exploit the use of these OPL words.

The exception is 'ESCAPE OFF/ON', which can be used to prevent programs from being paused by pressing **CLEAR/ON**, or stopped by pressing **CLEAR/ON**, **Q**. For normal applications, the use of these words is unnecessary: they are intended more to prevent a user breaking off half-way through a procedure when, say, only half of a set of data has been saved. This is more appropriate if someone else is going to use the program. The only routine in this book which could call for the use of 'ESCAPE OFF' is 'PASSWORD' (Program 3.6.1). However, when it is run, this program 'sits' waiting either for an 'INPUT' or to be switched on, and consequently it is virtually impossible to break into it (to examine the keyed-in Password) using the **CLEAR/ON**,**Q** technique.

If 'ESCAPE OFF' is used and Organiser II gets hung in a continuous loop, there is no escape. It is, therefore, a potentially dangerous instruction to use, particularly when developing and testing programs. The 'ESCAPE ON' instruction re-enables the escape process: this is the normal condition for Organiser II.

It is well beyond the scope of this book to explain the operation of Organiser II at the *machine code* level: as you will appreciate, the broad description of the inner workings given in Part 1 barely scratches the surface. For completeness, a description of the function of the *machine-level* instructions is given, with general examples of use where suitable. For interest, we will use some of the instructions to examine more closely how information is stored in Organiser, and to create a character pattern generator – with

facilities for saving the characters you create on a 'permanent' basis. However, it cannot be overstressed that to experiment with the instructions or to use them carelessly can be a dangerous exercise.

Locating variables in memory

For sophisticated programs, it can be useful to know the actual start address of where a variable is being saved in memory. The OPL word providing the answer is 'ADDR', which must be followed by the variable's name in brackets. Use of this instruction can be demonstrated in the CALCulator mode, to identify the addresses of the CALCulator memories M0 to M9.

Select CALC from the main Menu, then enter

ADDR(M0)

and press **EXE**. The first memory box allocated to 'M0' will be displayed (8447). If you now find the start address for CALCulator memory M1, you will see that there is a difference of eight memory boxes – the number required by Organiser to store *any* floating point number.

Examining the contents of memory

OPL has two words enabling the *actual* contents of memory to be examined. You will recall, from Part 1, that each memory box in Organiser can only store numbers from 0 to 255. The first word 'PEEKB' (short for 'PEEK Byte') examines the contents of *one* memory box. The address of the memory box to be examined must be included, in brackets, after the instruction: thus 'PEEKB(8447)'. This instruction can be used safely in the CALC mode of Organiser and, if you are inquisitive, you can examine just how Organiser stores floating point numbers. (But you will need to understand *hexadecimal* – and convert the contents of the first six addresses from decimal to hexadecimal values – and it would help if you understood *Binary Coded Decimal*. Another time, perhaps?).

The second OPL word is 'PEEKW' (short for 'PEEK Word'). In computer terminology, two bytes taken together are called a *word*. Each separate box can store from 0 to 255: if the second box is regared as a *counter* which pegs up one each time the first box reaches 255 (rather like the milometer on a car, but using different numbers), then the two together can be used to count from 0 to 65535. 'PEEKW(address)' gives the value of the contents of *two* consecutive memory boxes, starting at the address defined in brackets.

Two memory boxes are used together in the operating system to identify, for example, an *address* within Organiser II. Which explains (perhaps) why the highest accessible address in the system

is 65535 in decimal (or 'FFFF' in hexadecimal).

Some addresses used by Organiser II to store specific information are given in Appendix E of the Operating Manual supplied with the machine, together with the 'default' contents of those addresses (in hexadecimal). For example, using 'PEEKW($20CD)' in the CALCulator mode will reveal that the delay before Organiser II switches off automatically is 300 seconds ($12C in hexadecimal). The YEAR, MONTH, DAY, HOUR, MINUTES and SECONDS information is stored in memory boxes 8389 to 8394 respectively. 'PEEKB(8389)' for example will reveal the last two digits of the current year.

Changing memory contents

Just as there are two OPL words to *examine* the contents of one or two bytes of memory, so there are two OPL words to *change* the contents of memory boxes. These are danger words. Don't use them unless you know precisely what you are doing.

The first is 'POKEB' (short for 'POKE Byte), and it is followed by two parameters, separated by a comma. The first parameter specifies the address of the memory box concerned, and must lie within the range 0 to 65535, while the second parameter specifies the value to be stored at the designated address, and must lie in the range 0 to 255.

Addresses above 32767 must either be written in hexadecimal (and prefixed by a $ symbol), or written as the negative value resulting from subtracting 65536 from the address. Thus, address 65530 would be written as –6 (i.e., 65530–65536).

The second word is 'POKEW', and as with 'POKEB', it must be followed by two parameters for the address and the information to be stored in the specified memory. This time, however, the value to be stored is converted to hexadecimal and the two 'bytes' are stored at the two addresses starting with the designated address: the most significant byte is stored at the lower address.

Unless you really understand these words, don't 'poke' around without guidance. If you do know what you are doing, you can change some of Organiser's 'delays' to suit your own needs – the delay before Organiser automatically switches off, for example, or the delay before a pressed key 'auto-repeats'.

Defining your own characters

Two programs using POKEB are given in your Organiser Operating Manual: the first, called 'MUTE', enables you to switch off Organiser's sound system, while the second, called 'UDG' enables you to define the shape of eight of your own characters. These must be re-defined *each time Organiser is switched on.*

The characters are defined on a grid of five horizontal boxes by eight vertical boxes, as shown below:

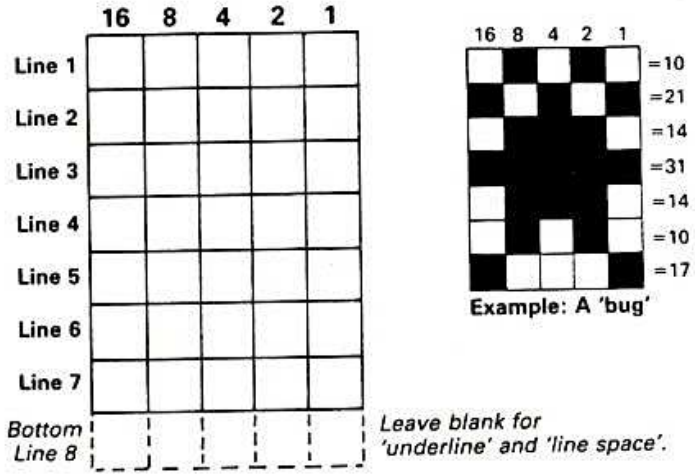


Fig. 3.18.1 Character definition grid

The complete character is stored in eight memory boxes, one memory box to a horizontal line. The contents of the memory boxes determine the actual pattern for that horizontal line. The box contents should be regarded as switches, turning 'on' a part of the pattern if they are 'switched on'. There are, in effect, eight switches to a box: only the *lower* five of these are used to define the shape.

Each 'switch' relates to a decimal value – as indicated in Fig. 3.18.1. (This is also discussed in Chapter 1.2, under the heading *Computers count differently*). Thus, if the top memory box is storing the value '4', then the switch associated with that value will be 'on', and a black dot will appear in the corresponding position in the top line of the character pattern. To set two (or more) switches 'on', simply add the corresponding decimal values together.

Thus, to create a solid black line at the very top of the character pattern, the first memory box must contain the value '31' (16 + 8 + 4 + 2 + 1).

The procedure for defining a character pattern is as follows:

a) Prepare the shape on a 5x8 grid, by shading in the required pattern.

b) Convert each horizontal line into a decimal value, by adding the values related to each vertical position.

c) Use the 'UDG' procedure from the Operating Manual or the 'CHRPAT' program that follows, to select the required pattern number and to enter the character pattern into Organiser II. Remember you can only define character pattern numbers 0 to 7.

d) To display the pattern in one of your own programs, use the instruction 'PRINT CHR$(number%)', where 'number%' represents the number of the pattern that has been defined.

Organiser does not retain the patterns when it is switched off. This can be an inconvenience if you wish to use specific shapes every time you run a program. Consequently, it can be useful to *save* the information defining a character pattern to a file. The following program suite enables you to define a character, to save it to a file, and to call that character back for loading into memory. The 'loading back' routine can be used as part of one of your own procedures to re-load previously defined characters.

The suite uses GLOBAL variables, and hence all procedures must be entered for the program to run. If you wish to use any of the procedures on its own, the relevant variables must be declared within the procedure in the usual manner.

First, here is the main controlling procedure:

```
CHRPAT:
GLOBAL CHR%,M%,HL%(8),F$(10)
START::
M%=MENU("NEWCHR,SAVECHR,LOADCHR,END")
IF M%=1
  CHRDEF:
ELSEIF M%=2
  CHRSAV:
  ONERR OFF                    :REM Put on in CHRSAV
ELSEIF M%=3
  CHRLD:
  ONERR OFF
ELSE RETURN
ENDIF
GOTO START::
```

Program 3.18.1a 'CHRPAT' Character definition control procedure

It will not be possible to test this procedure until the called procedures have been entered. The 'CHRDEF' procedure follows:

```
CHRDEF:
LOCAL C% :REM and CHR%, HL%(8) if stand-alone
CHRIN::
CLS
PRINT "CHARACTER (0-7)"
TRAP INPUT CHR%
IF (ERR) OR (CHR%<0) OR (CHR%>7)
  GOTO CHRIN::
ENDIF
C%=1
DO
  LINE::
  CLS
  PRINT "LINE ";C%
```



```
  TRAP INPUT HL%(C%)
  IF ERR
    GOTO LINE::
  ENDIF
  C%=C%+1
UNTIL C%=8
CLS
POKEB $180,64+(CHR% AND 7)*8   :REM Get this line right
C%=1
DO
  POKEB $181,HL%(C%)           :REM - and this one.
  C%=C%+1
UNTIL C%=8
PRINT "CHARACTER= ";CHR$(CHR%)
GET
```

Program 3.18.1b 'CHRDEF' Defining a character

Note that this procedure defines only the top *seven* lines of the character: the bottom line is usually left blank to separate the character from any displayed below it, and to allow for Organiser's underscore character.

Now here's the procedure to save a definition to a file:

```
CHRSAV:
LOCAL C%,T$(1)
ONERR BAD::
SAVE::
CLS
PRINT "FILENAME";CHR$(63)
TRAP INPUT F$
T$=UPPER$(LEFT$(F$,1))
IF F$=""                       :REM Escape route,press EXE
  RETURN
ELSEIF (T$<"A") OR (T$>"C") OR (MID$(F$,2,1)<>":")
  PRINT "INVALID LOCATION"
  GET
  GOTO SAVE::
ELSEIF EXIST (F$)
  PRINT "FILE EXISTS-NEW"
  PRINT "NAME PLEASE"
  GET
  GOTO SAVE::
ELSE CREATE F$:A,LINE%
  A.LINE%=0
  APPEND
  ERASE
  C%=1
  DO
```

```
    A.LINE%-HL%(C%)
    APPEND
    C%-C%+1
  UNTIL C%=8
  CLOSE                    :REM For more characters
  PRINT "CHARACTER SAVED"
  PRINT "TO ";F$
  GET
  RETURN                   :REM This line VITAL
ENDIF
BAD::
PRINT ERR$(ERR)
GET
C%=MENU("TRY-AGAIN,STOP")
IF C%=1
  GOTO SAVE::
ELSE RETURN
ENDIF
```

Program 3.18.1c 'CHRSAV' Saving defined character to a file.

Notice the use of error protection in this procedure – and the various escape routes included: pressing **EXE** when asked for the FILENAME will return you to the main procedure MENU. Invalid locations are 'trapped' (remember you must enter the location letter followed by a colon before the name itself). You are not allowed in this procedure to overwrite an existing file: if you wish to use the filename again for a new character shape, you must delete the original file first (using the File Management program in Chapter 16). All other errors are trapped by the 'ONERR' instruction – which displays the error and gives you the opportunity to break out of the program back to the main procedure MENU, or to try again from the start.

Finally, here is the procedure to load back a previously saved character. This procedure can be adapted to stand-alone for use in your own programs, if you wish: to do this, the variables used in the procedure must be declared.

```
CHRLD:
LOCAL C%,T$(1)
ONERR BAD::
START::
PRINT "FILE TO LOAD";CHR$(63)
INPUT F$
T$=UPPER$(LEFT$(F$,1))
IF T$=""
  RETURN
ELSEIF (T$<"A") OR (T$>"C") OR (MID$(F$,2,1)<>":")
  PRINT "INVALID LOCATION"
  GET
```



```
  GOTO START::
ELSEIF EXIST(F$)
  OPEN F$,A,LINE%
  C%-1
  FIRST
  WHICH::
  CLS
  PRINT "CHARACTER (0-7)"
  TRAP INPUT CHR%
  IF (ERR) OR (CHR%<0) OR (CHR%>7)
    GOTO WHICH::
  ENDIF
  POKEB $180,64+(CHR% AND 7)*8     :REM Take care
  DO
    POKEB $181,A.LINE%  :REM and this one too.
    NEXT
    C%-C%+1
  UNTIL C%=8
  CLOSE
  CLS
  PRINT "CHARACTER ";CHR%
  PRINT "NOW - ";CHR$(CHR%)
  GET
  RETURN
ELSE PRINT "FILE NOT FOUND"
  GET
  GOTO START::
ENDIF
RETURN                     :REM This line is VITAL
BAD::
PRINT ERR$(ERR)
GET
C%=MENU("TRY-AGAIN,STOP")
IF C%=1
  GOTO START::
ENDIF
```

Program 3.18.1d 'CHRLD' Load a presaved character pattern.

To use all of this program, run 'CHRPAT' first, select 'NEWCHR' to define a character pattern and then, if you wish, save the character information to a file by using the 'SAVECHR option. To load a previously saved character, select 'LOADCHR', and enter the file name containing the character pattern information that you want – remembering to include the location. You will then be asked which character number (from 0 to 7) you want the character information to be loaded into: thus you can design a range of characters, and call them back into whichever character numbers you wish.

After a character has been designed or loaded back into memory, the pattern is displayed on the screen so that you can check it is as you want.

If you wish to use the 'CHRLD' routine in one of your own programs – to set up character patterns, remember to declare the necessary variables – HL%(8), for example – at the beginning. Also you will probably choose to enter the file names as part of the program, rather than have them entered from the keyboard.

Machine language programs

Two OPL words – USR and USR$ – enable machine code programs to be run. Writing and entering such programs is beyond the scope of this book: indeed, it is unlikely that anyone except the most experienced would write machine code routines on Organiser. OPL is sufficiently powerful – and fast – to meet all but a few very highly specialised requirements.

Careless use of these OPL instructions can result in the loss or corruption of saved information and an upsetting of Organiser's operating system data. They are not discussed further.

A Final Word

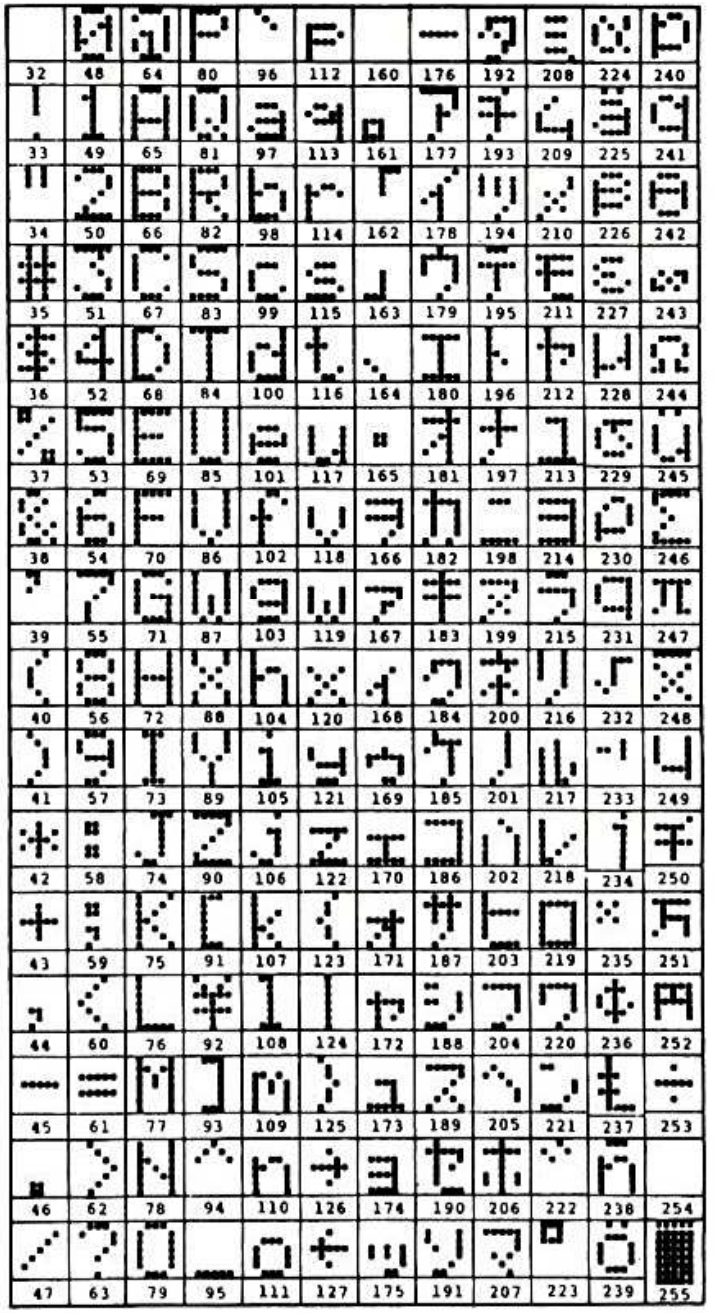
It is hoped that this book has given you an insight into the operation of your Organiser, and an understanding of how to program it to suit your own requirements. Remember, practice makes perfect.

Every effort has been made to ensure the accuracy of its contents, and to present the information in a way that can be readily assimilated by all. Suggestions for additional information or improvements should be sent to the Publishers for forwarding.

APPENDIX

ASCII – Character Pattern Chart

Note that patterns for characters 0 to 7 can be defined by the user – see Chapter 3.18.



INDEX

OPL commands and functions are identified by *OPL inst.*, and only the main descriptive entry for each is given. Generally speaking, this represents the first mention of the OPL word within the book. For program information, please refer to the Contents pages.

ABS *(OPL inst.)* 75
Adding MENU option 42
ADDR *(OPL inst.)* 245
ADDRESS
  Principle 2
  Next Instruction 19
ALARMs 47
  Cancelling 48
  Modifying 48
  Setting 47
AND *(OPL inst.)* 148
Animation 177
APPEND *(OPL inst.)* 204
ARRAY variables 105, 128
ASC *(OPL inst.)* 183
ASCII 14
  of control keys 166
Assigning values 116
AT *(OPL inst.)* 123
ATAN *(OPL inst.)* 75

BACK *(OPL inst.)* 215
BEEP *(OPL inst.)* 194
Binary system 11
Bit 11
BREAK *(OPL inst.)* 175
Byte 11

Calculation priorities 72
CALCULATOR
  Errors 77
  Principle 26
  Functions 74
  Memories 74, 110
  Using 66
Calling procedures 120
Central Processor Unit 3
CHARACTERS
  Control 13
  Principle 4
  Storing 12
CHR$ *(OPL inst.)* 132, 185
CLEAR/ON key 38
CLS *(OPL inst.)* 121
CLOSE *(OPL inst.)* 222
CONTINUE *(OPL inst.)* 175
Control key ASCII numbers 164
Converting variables 183
COPY *(OPL inst.)* 228
Copying files 55, 228
Copying procedures 98
COS *(OPL inst.)* 75
COUNT *(OPL inst.)* 218
Cursor keys 38
CURSOR ON/OFF *(OPL inst.)* 169
DATAPAKS
  Description 3, 9
Date setting 46
DATIM$ *(OPL inst.)* 190
Decorating Material program
  Planning 82
  Writing 154
DEG *(OPL inst.)* 75
DELETE *(OPL inst.)* 227
DELete key 39
Deleting
  DIARY entry 64
  MAIN records 54
  MENU option 43
DIARY
  FIND 61
  GOTO 61
  LIST 60
  Menu 58
  Saving 66
  TIDY 62
  PAGE 59
  Using 62
DIR$ *(OPL inst.)* 224
DIRectory of procedures 98
DISP *(OPL inst.)* 219
DO/UNTIL *(OPL inst.)* 174

EDIT *(OPL inst.)* 210
EDIT (PROGram MENU) 93
Editing MAIN records 53
ELSE/ELSEIF/ENDIF see IF
ENDWH see WHILE
EOF *(OPL inst.)* 215
EPROM 10
ERASE MAIN records 54
ERASE procedures 110
ERASE *(OPL inst.)* 220
ERROR handling 111, 237
Error messages 233
Error trapping 237
ESCAPE *(OPL inst.)* 244
EXEcute key 40
EXIST *(OPL inst.)* 205
EXP *(OPL inst.)* 75

FILES, built-in 49
FILES, programmed 201
  Closing 222
  Creating 202
  Deleting 227
  Management 224
  Opening 204

FIND – DIARY 61
FIND – main MENU
  Principle 23
  Using 52
FIND *(OPL inst.)* 215
FIRST *(OPL inst.)* 214
FIX 69
Fixing decimal point 69
FIX$ *(OPL inst.)* 185
Floating point numbers 15
Floating point variables 104
Flowcharting 84
FLT *(OPL inst.)* 187
FREE *(OPL inst.)* 193
Frequency 194
FUNCTIONS
  Numeric 74
GEN$ *(OPL inst.)* 186
GET *(OPL inst.)* 123, 163
GET$ *(OPL inst.)* 163
GLOBAL variables
  Description 108
GOTO
  DIARY 61
  (OPL inst.) 153

HEX$ *(OPL inst.)* 187
HOUR *(OPL inst.)* 190

IF/ELSEIF/ELSE/ENDIF
  (OPL inst.) 143, 146
Information storage 50
INFO option 44
INPUT *(OPL inst.)* 122, 163
Instructions
  Separating 102
  types 101
INT *(OPL inst.)* 157, 188
Integers numbers 15
Integer variables 104
Interrupts 20
INTF *(OPL inst.)* 188

KEY *(OPL inst.)* 164
KEY$ *(OPL inst.)* 164
Keyboard
  Principle 4, 36
  Using 36
KSTAT *(OPL inst.)* 168

Labels 153
LAST *(OPL inst.)* 215
LEFT$ *(OPL inst.)* 134
LEN *(OPL inst.)* 136
LIST
  DIARY 60
LN *(OPL inst.)* 75
LOC *(OPL inst.)* 138
LOCAL variables
  Description 107
LOG *(OPL inst.)* 75
Loops 175, 242
LOW BATTERY 3
MEMORY 7
  Boxes 2
  Calculator 82, 106
  Content 10
  EPROM 10
  RAM 8
  ROM 7
MENU – DIARY 58
MENU – main
  Customizsing 42
  Description 41
  Selecting option 42
MENU *(OPL inst.)* 152
MENU – PROGram 91
  COPY 98
  DIR 98
  EDIT 93
  ERASE 97
  LIST 100
  NEW 92
  RUN 96
MENU – PROG SAVE 96
MID$ *(OPL inst.)* 134
MINUTE *(OPL inst.)* 191
MODE key 39
MONTH *(OPL inst.)* 191
Multiplying 68
Music 194
Nesting 145, 175
NEW (PROGram MENU) 91
NEXT *(OPL inst.)* 215
NUM$ *(OPL inst.)* 186
NUM key 37

OFF *(OPL inst.)* 129
ONERR *(OPL inst.)* 237
OPTIONS (main MENU) 42
  Adding 43
  Deleting 43
  Moving 43
  Restoring 43
  Selecting 42
OR *(OPL inst.)* 148

PAGE option 59
Parameters
  Description 109
Pattern numbers 13
  (See also 'ASCII')
PAUSE *(OPL inst.)* 165
PEEKB *(OPL inst.)* 245
PEEKW *(OPL inst.)* 245
PI *(OPL inst.)* 75, 193
POKEB *(OPL inst.)* 246
POSITION *(OPL inst.)* 217
POWER SUPPLY 3
PRINT *(OPL inst.)* 121
PROCEDURES
  Calling 120
  Creating 113
  Declaring variables 115
  Naming 113
  Principle 30
  Ways to run 31
Program Counter 19
PROGRAM MENU (See under MENU)

PROGRAMMING
Flowcharting 84
Introduction 101
MENU 91
Planning 82
Principles 83

QUIT (PROG Save MENU) 96

RAD *(OPL inst.)* 75
RAM 8
RANDOMIZE *(OPL inst.)* 193
Records (main MENU)
Editing 53
Keeping 49
Saving 51
Records (Programmed)
Adding 207
Changing 208
Displaying 218
Editing 210
Fields 203
Finding 214
Selecting 212
Size 211
RECSIZE *(OPL inst.)* 211
RENAME *(OPL inst.)* 229
REM *(OPL inst.)* 177
RESET option 44
RETURN *(OPL inst.)* 118
RIGHT$ *(OPL inst.)* 134
RND *(OPL inst.)* 75, 129
ROM 7
RUN (PROGram MENU) 96

SAVE (main MENU)
Principle 23
Records 51
SAVE (PROG Save MENU) 96
SCI$ *(OPL inst.)* 187
Scientific notation 71, 187
SECOND *(OPL inst.)* 191
SHIFT key 37
SIN *(OPL inst.)* 75
Sound effects 198
SPACE *(OPL inst.)* 226
SQR *(OPL inst.)* 75
STOP *(OPL inst.)* 241
STORING
Characters 12
Cutting 127
Instructions 14
Values 15
STRINGS 105
Arrays 128
Concatenating 131
Converting 185
Declaring 127
Variables 128
Switching off 45
Switching on 41

TAN *(OPL inst.)* 75
Test operators 137
TIME setting 46
Translation service 18
TRAN (PROG Save MENU) 95
TRAP *(OPL inst.)* 239

UNTIL see DO
UPDATE *(OPL inst.)* 208
USE *(OPL inst.)* 212
USR *(OPL inst.)* 252
USR$ *(OPL inst.)* 252

VAL *(OPL inst.)* 184
Variables
Array 105
Assignments 116
Converting 183
Declaring 115, 127
Description 103
Floating point 104
GLOBAL 107
Integer 104
LOCAL 107
Names 106
Non-declared 110
Parameters 109
String 127
VIEW *(OPL inst.)* 170, 218

Warehouse concept 2
Office 3
WHILE/ENDWH *(OPL inst.)* 173

YEAR *(OPL inst.)* 191

This book contains a detailed explanation of how to use the Organiser II. Many examples of its procedures and programs are included in a style that is both easy to follow and entertaining. By taking the time to understand this very powerful device, its full potential can be realised. The detailed description of the Organiser II, its built in applications and programming capability will make light and enjoyable the task of exploring this fascinating computer, particularly for those users who have not used such device before.

".....this book can be read with benefit by the expert and uninitiated alike..."
David Potter Chairman Psion Ltd.

£9.95

Published by
Kuma Computers Ltd., Pangbourne, Berkshire, England
Telex: 846741 KUMA G. Telephone:07357-4335

Kuma

