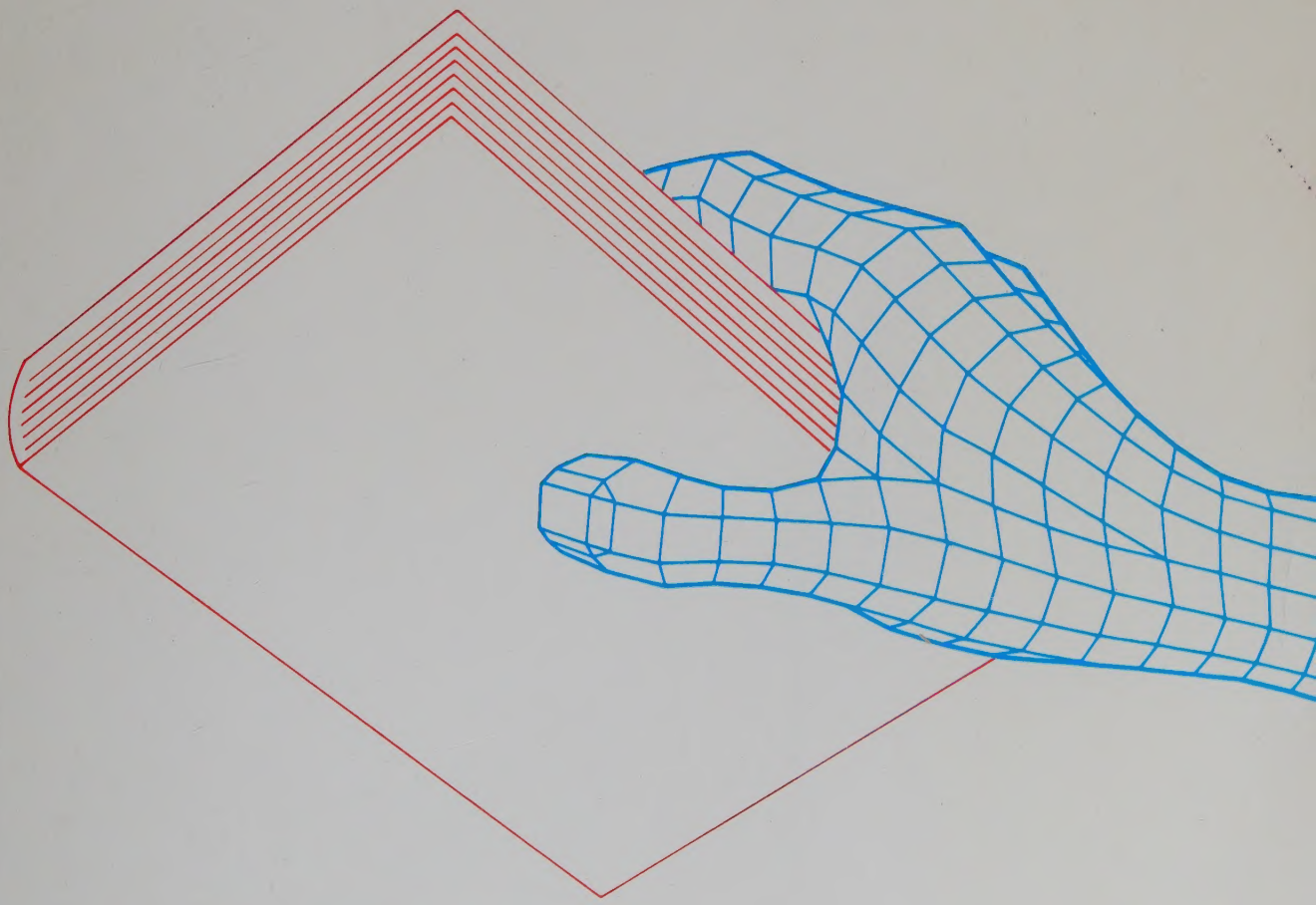
QL Archive



Ian Murray of *BLUEPRINT*

QL Archive

a *BLUEPRINT* handbook

Ian Murray



Century Communications Ltd.

Copyright © Blueprint (Spottiswoode & Spottiswoode Ltd) 1985

All rights reserved

First published in Great Britain by
Century Communications Limited
(a subsidiary of Century Publishing Co. Ltd)
Portland House, 12–13 Greek Street, London W1V 5LE

ISBN 0 7126 0633 5

Originated directly from the author's word-processor disks by
NWL Editorial Services, Langport, Somerset, TA10 9DG

Printed in Great Britain by
Butler & Tanner
Selwood Printing Works,
Frome,
Somerset

The ARCHIVE handbook

The ARCHIVE handbook is an excellent, straightforward guide for people who are unfamiliar with the use of databases. It also has much to offer the sophisticated user.

Written without jargon, the book explains how to use ARCHIVE quickly, effectively and confidently. In addition to teaching basic programming skills it includes:

- Step-by-step instructions
- Helpful hints
- How to avoid making simple mistakes
- How ARCHIVE works (and what to do when it doesn't)
- Quick reference section
- Glossary
- Programming hints
- Exercises and suggested answers

Every example is explained both in detail and in brief, letting you set your own pace.

The author has had extensive practical and technical experience with microcomputers, and has written a clear, informative guide to using databases, programming and QL ARCHIVE.

About the Author

Ian Murray is managing director of Synchronicity Ltd, a company which specialises in microcomputer software development and in teaching people how to use computers. He read English Literature at Cambridge and then worked for several years in the computer side of the publishing industry. After this, he worked as a freelance computer consultant before setting up his own company.

ACKNOWLEDGEMENTS

A special thank you to Caroline Atkinson for her invaluable assistance. It is hard to imagine how this book could ever have been completed without her. Thanks also to Steve Di Biasio, who sat down at a computer for the first time to go through the examples and edit and check the text.

I would also like to thank my publisher, Richard Brockbank, for his useful insights and patience, and I am grateful to Charles Davies of Psion, author of ARCHIVE, for all his help.

Finally, I would like to thank Francesca Simon and Clare Spottiswoode for the Blake family, the series style, and for leading the way.

Table of Contents

CHAPTER 1 – First Steps 1

Getting out of trouble 1
The QL 1
The Keyboard 2
Microdrive cartridge 3
Labelling cartridges 3, Protecting a cartridge 3, General care 4
Copying the program cartridge 5
Formatting a Microdrive cartridge 6
Starting up ARCHIVE 7
Starting ARCHIVE from the basic screen 9
Display widths 9
The Screen 10
Control area 10, Work area 10, Display area 11
Notes on the screen 11
Using the function keys 12
F1: The HELP key 12, F2: Turning the control area on and off 13, F3: The commands list 13, F4 14, F5 14
Editing text in ARCHIVE 14
Entering and editing lines 14, Erasing text 15, Inserting text 15, Moving within the text 15
Keys used in editing – the Line Editor 16
Leaving ARCHIVE 16
QUIT command 16

CHAPTER 2 – The First File 17

Example: Miranda Organises Her Wedding

Introduction 17
Creating a file 18
Instructions 19
Inserting information into the file 20
Leaving INSERT 22, Displaying information 22
Moving around the file 23
The BACK command 23, The LAST, NEXT and FIRST commands 23
Adding notes to the memos file 24
Finding and altering records 25
Altering records 26, Finding and altering more records 26
Saving the file 28
Listing the files on tape – the DIR command 29, Closing files 29
Summary 31
Helpful hints 32

CHAPTER 3 – Multiple Files 33

Examples: The Guest List
Using a Logical Name with a Command

Points to remember 33
The second file 34
Getting started 36
The guests 36
Creating the file 36, Displaying the structure 39, Looking at the directory 39, Changing the current file: Logical name 39
Using multiple files 41
Displaying the current file 41, Adding notes to the `memos` file 41, Deleting records 42, Adding records to the `guests` file 43, Changing the current file with `USE` 43, Editing a field 45, Entering text in number fields 46, Recovering from errors in `INSERT` 46, Saving the record 47, Abandoning the record 48, Adding more guests 48, Displaying the last record added 50, `INSERT` and `ALTER` 50, Closing multiple files 51
Making backup copies of files 53
Copying the same tape with `BACKUP` 53, Formatting a cartridge 54, Copying to another drive 55, Rotating backups 55, Dating your cartridge 55
Summary 57
Helpful hints 57

CHAPTER 4 – Functions and Procedures 59

Example: Using the Guest List

Points to remember 59
Getting started 60
Counting the file 60
The `PRINT` command and the `COUNT()` function 60, Displaying the results of `COUNT()` 61, Displaying the result as a decimal fraction 62, Displaying the result as a percentage 62, Displaying the cost of postage 62
Looking at the guests 63
Restoring the display – the `SCREEN` command 63, The standard display layout 64, Browsing through the file 65
Creating commands (procedures) 65
Creating a procedure to move forwards through the file 65, Using your own procedures 67, A second procedure to move backwards through the file 68, Moving to another procedure 70, Creating more procedures 71, Saving copies of procedures – the `SAVE` command 72

A starting procedure to open files 73
Open guests file and memos file 73, A procedure to open both files 74, Procedures to close each file 75, A procedure to close both files 76, A procedure to backup both files 76, A backup procedure 77
Putting your procedures together 78
The START procedure and the RUN command 78, Bringing procedures into current memory – the MERGE command 78, Saving procedures 79, Making a backup copy 79, The procedures in wedfiles_prg 79
A general purpose backup 81
The INPUT command 81, Using a variable 82, Adding strings of characters 84, Messages and reminders 84
Saving and loading procedures 85
Loading procedures – the LOAD command 85, Notes on saving procedures 86, A procedure to clear procedures but not data 86, Using the DUMMY procedure file 86, Notes on saving and loading procedures 87
Summary 88
Helpful hints 88

CHAPTER 5 – Loops 89

Example: Joyce Takes Over – Fast Forward

Points to remember 89
Starting up 90
A procedure to move quickly through the file 91
Instructions 91, The ALL command 91, Using ALL in a procedure 91, Loops 93, Displaying the record number 93, A procedure to display the record number 93, The RECNUM() function 94, Using the PRINREC procedure 95, The WHILE loop 95, Using the amended ZOOM procedure 96
Tracing mistakes 96
Turning off TRACE 98, Editing PRINREC 99, Using ZOOM with another file 100
Creating a procedure to list the file 100
Replicating a procedure 101, Using the listing procedure 102, Printing the list on paper 103, Multiple commands on one line 103, Saving the procedures 103

Adding fields to a file 104
Creating the new file structure 104, Appending records from the old file to the new 104, The LET command 106, Polishing the TRANSFER procedure 106, Renaming a file 107, Deleting a file with KILL 107, Changing all records 108, Updating records 108, ARCHIVE and dates 111, Adding the data 111, Altering each record in the file 111
Summary 113
Helpful hints 113

CHAPTER 6 – Listing and Printing Information 115

Example: Printing Lists

Points to remember 115
A procedure to list the memos file 116
Instructions 116, Carriage returns 117, Final semi-colons and the PRINT command 119. Tabs 119
Adding colour – Nigel gets fancy 121
The PAPER and INK commands 121, Print items 123, Using VAL() to get numbers from strings 124
Printing on a printer 125
The LPRINT command 125, On line 127, Installing printers 127, The printer_dat file 128, Printing the memos 128, Form feeds 129, The CHR() function 129
Listing addresses 130
Changing the listing procedure to list the guests 130, Passing parameters 132, Using parameters 132, Saving the procedures 135, Editing procedures – cut and paste 135, Cut 135, Paste 137, Using the FIELDV() function 138
A standard listing procedure 138
How the procedure works 139, Using the standard listing procedure 140, The function NUMFLD() 141, Printing out procedures with LLIST 142, Deleting outdated program files 143
Summary 144
Helpful hints 145

CHAPTER 7 – Selecting Information 147

Example: Wedding Preparations – What and When?

Points to remember 148
Starting up 148
Finding specific records 149
Instructions 149, The SEARCH command 149, The functions UPPER() and LOWER() 150

A procedure to keep finding specified records 150
Deliberate forever loops 150, Matching an empty field with a null string 151, The IF command 152, The function FOUND() 152, IF and ENDIF 153, True and false 153, Using the SEEMORE procedure 155, Greater than and less than 155, Improving SEEMORE 157, Finding part of a field with the string slice 157, Finding ranges of records 158, Searching strings with the INSTR() function 159, The SELECT command 160, Resetting the file 162
Further selections 162
Selecting with AND 162, Selecting with OR 164
Checking data with logical expressions 165
A range-checking procedure 165, Two possibles 166, Several different possibles 166, Checking dates 167, Reversing the logical outcome 167, Notes on evaluating logical operators 168, Fully worked examples 168, Example 1: OR 169, Example 2: AND 169, Example 3: OR, AND 169, Example 4: (), OR, AND 170, Avoiding logical operators 170, Tidying up the tape 171
Summary 171
Helpful hints 172

CHAPTER 8 – Sorting Information 173

Example: Putting the Guests in Order

Points to remember 173
Starting up 174
Sorting and sequencing 174
The ORDER command 174
Automatic ordering 176
The LOCATE command 177
Ordering numeric fields 177
Points to remember 177
Ordering text fields 179, Ordering by date 179, Ordering on more than one field 181, Secondary sorts 182, Composite keys 182, Saving the order 183, How ORDER works 183, Tidying up the tape 183
Summary 184
Helpful hints 184

CHAPTER 9 – Maintaining Records 185

Example: RSVP

Starting up 186
Procedures to log expected guests 186
A procedure to get a one-letter command 187, The GETKEY() function 188, Running the procedure COMGET 188, Naming procedures 189, A procedure to act on keys pressed 189, Amending the COMGET procedure 190, Menus 191, Using COMGET with the guests file 191, Providing a cursor for GETKEY() 192, Preventing errors from stopping a program 193, Flowcharts 194, The ERROR command 194, A procedure to close all open files 194, Running CLOSEALL 195, The ERRNUM() function 195, Error reporting 196, Using ERROR and ERRNUM() 196, Creating a new starting procedure 198, Identifying programs and procedures 199
Finding the right people 200
Nested IFs 201, The ELSE command 201, Adding new options to COMCHECK 202, Altering the SEEMORE procedure 202, The RETURN command 203, Using the program to find some names 203, A procedure to clear unnecessary lines from the screen 204, The function REPT() 204, Inserting BLANK into other procedures 205, Standard prompts and warnings 207, Converting numbers to strings 209, The STR() function 209, Using the number-logging procedures 210, Logging the first reply 211, A procedure to help identify the command letters 211
Exercises 215
Suggested answers 217
Help 220
Saving your work 220
Summary 221

CHAPTER 10 – Screen Design 223

Points to remember 223
Designing your own screen 223
The SEDIT command 224, Creating a screen display using SEDIT 225, The screen title 225, Using colour in SEDIT 225, Setting the first variable 226, Setting second variable 227, Setting more variables 227, Saving screen files – the SSAVE command 229, Coping with mistakes in SEDIT 229
Using your own screen 230
Loading screen files – the SLOAD command 230

Adding the new screen display to your procedures 231
Screen files summary 232, Incompatible versions of screen files 232
Summary 233
Helpful hints 233

CHAPTER 11 – A New Program from Old Building Blocks 235

Example: The Toy Charts

Points to remember 235
Instructions 236
Multiple one-to-many relationships 237, Starting up 238, The `toys` file 238, The `sales` file 239, The toy sales screen layout 240
Converting `wedcom` for toy sales 242
Examining the task 242, Getting started 242, Changing the current file 244, Linking the files 246, The LINKRECS procedure 247, Deciding which file to link 247, Linking `toys` with `sales` 248, Linking `sales` with `toys` 249, Some records to get started 252, The `SINPUT` command 253, Inserting a new toy 253, Prompt for a new code 255, Clear the fields for input 255, Input values for the `toys` file 256, Tidying up 257, Inserting and calculating a new sale 257, The calculation 259, Operations priorities 260, Sales of Larry Lizard 261
Summary 262
Helpful hints 263
Program listing 263
Exercises 268
Suggested answers 269

CHAPTER 12 – Listing, Paging and Totalling 271

Example: Listing the Toy Sales

Points to remember 272
Instructions 272
List on screen or printer? 272, Changing the `DISPREC` procedure 273, The procedure `LINECR` 274, Changing `LIST` 274, The function `FIELDT()` 275, Pausing the listing – first count the lines 277, Taking action when the screen or page is full 278, Declaring temporary working values 280, Totalling the profits 281, Subtotalling by toy 282, Listing and totalling sales for one toy 283, Averaging the sales 285, Local variables 286, Linking the files 287, Merging the two programs 289, Selecting and listing a range of records 291, Formatting numeric output 293, The function `GEN()` 294
Summary 295
Helpful hint 295
Program listing 295

CHAPTER 13 – Output Plain and Fancy 299

Examples: Harold Puts the Boot In
Labels and Another Way of Printing Files

Points to remember 299
The _scr file 301, Exporting to QUILL 301, Exporting to EASEL and ABACUS 302, The common format 302, Exporting sales to EASEL 302

Changing file structures 305
The common format in detail 306, Printing elsewhere with SPOOLON 306, Import errors 309

Instructions 309
The DUMP command 310, Nigel enhances the dump 310, Computer label stationery 312, Using loops to process labels 314, The label line file 314, Vital statistics 315, Storing a row 316, Storing a line 317, Printing the row 317, The controlling procedure 319, Test print 320, A program menu 321, How to use the program menu 323, Arrays and string handling 323, Strings as fixed length arrays 324, Slicing variable length substrings 325, Slicing address lines 326, More about blank address fields 327, Folding long lines 328, Numbered delimiters 328, Procedures which write procedures 330, Complex and varying selections 331, Object files 333

Helpful hints 333
Listing of label printing program 334

APPENDIX A – Getting Out of Trouble 337

APPENDIX B – Memory and Maximum Records 345

APPENDIX C – Differences Between Versions 348

APPENDIX D – ASCII Codes 349

INDEX 351

First Steps

You may already be familiar with much of the contents of this chapter if you have read *First Steps* in any of the other Blueprint books on the QL programs. Some information is new and specifically relates to ARCHIVE – check the sections called *The Screen*, *Commands* and *Editing Text*.

Getting out of trouble

Leaving ARCHIVE is described at the end of this chapter, and the use of the `reset` button on the next page. If the computer stops responding, or if you encounter strange messages, consult the appendix *Getting Out of Trouble*.

THE QL

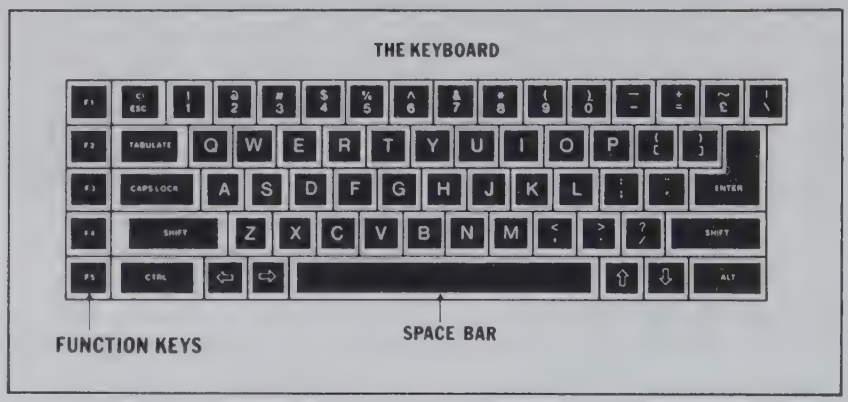
The slim Sinclair keyboard holds a sophisticated computer. It is a powerful machine, waiting to do work and to obey instructions. Software is the program of instructions that, with your help, tells the computer what to do. ARCHIVE, QUILL, ABACUS and EASEL, the programs that come with your QL, are all software. The software is recorded on the Sinclair cartridge, rather like the music on an audio cassette.

The main difference is that you can switch off a recorder at any time without harming the tape. If you pull out the plug on a computer without recording the work you've created, it will be lost. Make sure you save your work on tape regularly, and make copies of it on more than one tape.

Set up your computer with screen, keyboard, and power connected (see appropriate manuals). Now look at the keyboard. It resembles a conventional typewriter; the main difference is that there are a few more keys.

KEYBOARD

The keyboard is shown below, with an explanation of what all the keys do. All of them will be explained in detail in this book.



[↵] Tells ARCHIVE that you've finished typing a line

[ESC] **ESCAPE** key. Can be used at any time to return to the main screen, or to abandon a line

[CTRL] **CONTROL** key. Used in association with other keys to perform functions like deleting lines of text

[ALT] Ignore. Has no function in ARCHIVE

[SHIFT] Hold down the **SHIFT** key at the same time as pressing any other to convert letters into capital letters, or to print the upper character shown on a key.

[CAPS LOCK] Locks keys into capital letters when you press it the first time. Press it again to restore lower case letters. **CAPS LOCK** only affects letters.

[TAB] **TABULATE** key. Used to move up and down through fields in a record, or procedures in a program.

[RESET] Not a key in the normal sense, but a button underneath the right-hand edge of the QL. It has almost the same effect as switching the power off and on again. Only used when all else fails to escape from situation where the program has 'hung' or 'locked'.

ARROW keys:

[←][→] moves cursor to extreme left/right of a line

[↑][↓] moves cursor left/right

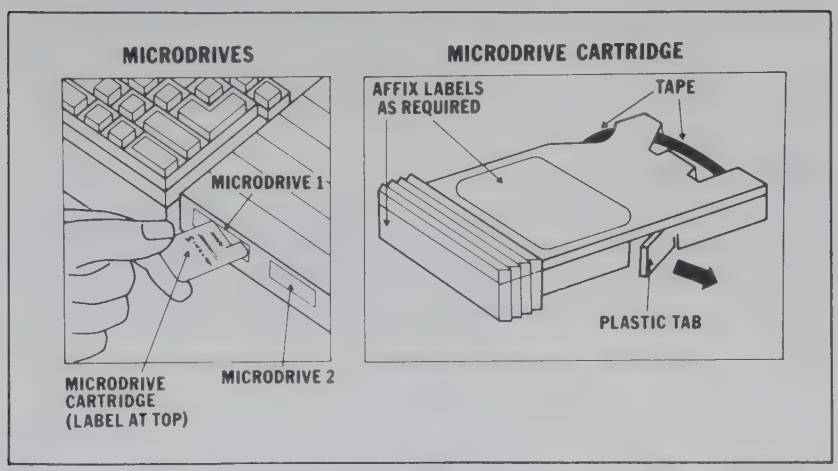
FUNCTION keys:

[F1] The HELP key
[F2] Switches CONTROL AREA on/off
[F3] Pages through list of commands in CONTROL AREA
[F4] Abandon `alter/insert` and `insert` in `edit`
[F5] The `line edit` key

MICRODRIVE CARTRIDGE

The cartridge is the black rectangular object that you insert into the Microdrive.

A Microdrive cartridge is similar to a cassette tape, but it has been specially designed for the needs of the QL computer. One of your cartridges holds the ARCHIVE program – it is in the special pack with EASEL, ABACUS and QUILL. You will need several blank cartridges to record your information and programs – like cassette tapes you can always re-record them later.



Labelling cartridges

There is a small rectangle on one side of the cartridge for the label (*see diagram*). It is important to keep this label up to date so you can see at a glance what is on the cartridge. When you place the cartridge in the Microdrive this label will be on the top face.

The label on a Microdrive cartridge is small. It is useful to label the cartridges with a number, and distinguish between *program* and *work* (data) cartridges. Keep a separate record of what is on each cartridge in a book. Keep a separate page for each cartridge, and as you add or delete files write the changes into the book. If you do not do this, you will be surprised how quickly you lose track of your work.

Protecting a cartridge

Look at the difference between your ARCHIVE cartridge and a new cartridge. The small projection along one side is missing on the ARCHIVE program cartridge. This is a neat safety device. Once this projection has been removed the computer will not add to or erase anything from that cartridge; it will only read information from it.

This means that neither you nor anyone else can accidentally erase your ARCHIVE program, or do anything to harm it by pressing the wrong keys.

If you later need to re-record the tape, the missing projection can be replaced by a small piece of tape or label placed across the opening. Make sure it is neatly and firmly in place, so that it does not come unstuck inside the drive.

Never remove the cartridge while the red light beside the drive is on. The program cartridge may be removed as soon as ARCHIVE has finished loading, as long as you do not intend to use ARCHIVE's help facility. Data cartridges can be freely interchanged as long as there is no program running which is making use of them, and any open data files have been closed. This is more fully discussed in Chapter 3. If you do remove the cartridge at the wrong time you could destroy your stored work or the cartridge.

General care

- Never touch the tape with your fingers, or insert anything into the cartridge.
- Always store the cartridge in its sleeve when not in use.
- Never move the QL with a cartridge installed – even if not in operation.
- Never touch a cartridge with the Microdrive running, (ie when the red light is on).
- If loops of tape appear at either end of the two places shown in the diagram, gently ease the loop back into the cartridge. Use a clean, smooth instrument for this, such as a pen or pencil. Never touch the tape with your fingers, for this or any other reason.
- Microdrive cartridges will not last forever, and will eventually need to be replaced. It is very important to keep back-up copies of precious programs and files on another cartridge.
- Protect your programs and data which you want to prevent from being over-written or erased by snapping off the plastic tab on the cartridge, as shown in the diagram.
- If you wish to overwrite a cartridge that has been protected, stick some tape over the place where the tab has been removed.
- Do not press the reset button on the side of the computer with the cartridges still in the Microdrives. Remove the cartridges before resetting. If the computer has 'hung up' and the Microdrive light is on, leave the cartridge in when you press the RESET button.

COPYING THE PROGRAM CARTRIDGE

Before using ARCHIVE for the first time you must make a copy of the program for safe keeping. This is vital. Microdrive cartridges don't last for ever, and can be damaged – quite apart from the normal hazards of wear and tear, the dog could chew the cartridge or a child could pull out the tape. Tripping over a power cable and breaking the power supply could also damage the program cartridge. In all these events you could damage, and thus lose, the stored program, ARCHIVE, on the cartridge.

To make a copy of the ARCHIVE program you need to follow certain procedures. Follow the instructions below exactly; it doesn't matter if you don't understand them. The process uses the QL's built-in computer language known as SUPERBASIC.

Afterwards you can forget all about it – you won't use it in ARCHIVE.

1 Switch the computer on
Ensure that the power cable is connected and power is on (the yellow light on the left of the QL confirms this). Don't insert any Microdrive cartridges into the machine just yet. After a few moments the screen will display the message:

```
F1..........Monitor
F2..........TV

©  1983  Sinclair Research Ltd.
```

Note: If you are using a TV you will need to tune it to the right channel for the QL. If you have enough channels, keep one of the spare ones permanently tuned for the QL.

F1 or F2

2 Press **F1** if you have a special computer monitor
F2 if you have a domestic TV

The computer will first check whether you have a program cartridge in the left-hand drive, drive 1, and as you don't you are presented with the QL BASIC SCREEN (a blank screen on a TV). The cursor (a small, light rectangle) blinks at the bottom of the screen waiting for your next move.

3 Place the ARCHIVE cartridge in the right-hand drive, drive 2

4 Place a new blank cartridge in the left-hand drive

`lrun mdv2_clone`

5 Type `lrun mdv2_clone`
Note that this is the letter 'l' not the *number* 1. Note also the blank space between `lrun` and `mdv2` (press the **SPACE BAR**) and the underline '_' (**SHIFT** + `hyphen` key) before `clone`. Trans-

lated, you are asking the computer to load and run a `basic` program called `clone` from the second (right-hand) Microdrive cartridge, (your ARCHIVE master cartridge).

If you make a mistake while typing, pressing the CTRL key at the same time as the left arrow key will delete one character to the left of the cursor.



6 Press ENTER
The Microdrives whirr into action. The QL responds '`Format mdv1 -- press space bar to continue`'

Note: 'Format' is a computer term which means the process of marking the Microdrive cartridge in such a way that the QL can find its way quickly around the tape. Be warned – this process of marking the cartridge erases anything that was previously on that tape. But for the moment we assume that you are using a blank tape.



7 Press SPACE BAR
The Microdrives whirr again, 'formatting' the cartridge in the left-hand drive. The ARCHIVE program cartridge is then copied file by file onto the new cartridge. The process takes some minutes, and once finished the cursor – which marks the place where you type in characters – will return to the bottom of the screen.

Note: The computer will not allow the ARCHIVE program cartridge to be reformatted since it has been protected by the breaking of its tab. This is an essential safety device which stops you from accidentally erasing ARCHIVE.

8 When the copy has been made (when the Microdrives stop whirring), *remove the original ARCHIVE program cartridge* (from the right-hand drive) and store it with your master EASEL, ABACUS and QUILL cartridges. Take your new working copy of ARCHIVE out of the left-hand drive, and label and date the cover so that you know what is on that cartridge.

Note: From now on always use your copy of ARCHIVE, *not* the original. This ensures that if anything does go wrong you still have an unscarred original program cartridge.

At some later point, when you have checked that your new copy of ARCHIVE works and your printer is installed correctly, remove the tab on the left of the cartridge. This will protect the cartridge, preventing you accidentally overwriting your copy of ARCHIVE.

FORMATTING A MICRODRIVE CARTRIDGE

To use ARCHIVE in the next section, you will need your copy of ARCHIVE and a blank, formatted cartridge. This blank cartridge will be used to store the files that you create.

If you do not have a formatted Microdrive cartridge, first ensure that you have the QL BASIC SCREEN in front of you (remember this is the red/white blank display with a flashing cursor at the bottom). Then follow these instructions:

1 Place a new Microdrive cartridge in the right-hand drive.

```
format mdv2_arcdata
```

2 Type `format mdv2_arcdata`
This uses the `format` command built in to the QL and asks the QL to format the cartridge in drive 2, the right-hand Microdrive, and to label that whole cartridge `arcdata`. Note the '_' and the space. Each time you format a new cartridge you have to give it a label (or name). This label can be up to 10 characters long, and should not include any spaces.

It is useful to give each cartridge a label (or 'volume name' as it is also known, as if tapes were books) which describes its purpose. A backup copy of a data tape might be labelled '`arcdata2`', for example.



3 Press **ENTER**
to execute the command. The right-hand drive will whirr for a few seconds, and once complete you are returned to the BASIC SCREEN.

The number of free sectors (in other words, the amount of free space on the tape), appears on the screen, eg. `218/219 sectors`. This means that the tape has 219 sectors of which 218 are free for use.

You can also issue the `format` command (see chapter 3) from within ARCHIVE itself. Be very careful when formatting tapes as anything already on the tape is wiped out by the formatting process. It is a good idea to have only the one tape in the drives, to ensure that you do not accidentally erase a tape in the wrong drive.

STARTING UP ARCHIVE

You are now ready to use ARCHIVE. First, the program must be 'loaded' into the computer. You will need your working copy of ARCHIVE and a blank formatted cartridge. If your QL is switched on and you are looking at the BASIC SCREEN with the cursor, move on to the next section. Otherwise,

1 Switch the computer on.
After a few moments the screen will display the message:

```
F1.....Monitor
F2.....TV

© 1983 Sinclair Research Ltd.
```

2 Insert your copy of the ARCHIVE cartridge in the left-hand Microdrive (drive 1).

F1 or F2 3 Press F1 if you have a monitor

F2 if you are using a TV screen

Look at the diagram of the keyboard if you are not sure which key to press. Pressing F1 tells the QL to use 80 columns to display its information, while pressing F2 instructs it to use a 64-column display.

After a few seconds the screen will clear, and show a message such as:

```
Loading Q L ARCHIVE
     Copyright
     Psion Ltd.

©    1984
```

The screen clears again and displays a further message giving the version number of your ARCHIVE program. The version number indicates how recent that version of the program is. Programs are like books, and are continually revised and republished in new editions or versions. The higher the version number, the more recent it is.

In this book, differences between version 1 and version 2 (Spring 1985) are pointed out where relevant.

Note: As a QLUB member, you will receive any updated versions of the *Psion* software packages automatically at least once a year. Any new version that you receive will contain corrections of known errors and also any improvements that have been made to the software. Any new versions of the programs are sent to QLUB members by Sinclair, not *Psion*.

The following message also appears:

```
WHEN USING DATA FILES, A FORMATTED MICRODRIVE CARTRIDGE SHOULD
ALWAYS BE PRESENT IN DRIVE 2
```

This simply means that data should not be stored on the program cartridge (in drive 1), and ARCHIVE assumes you will be using drive 2 for storing data.

This is because there is very little room left on the program tape, which may also have had its write-protect tab removed. You can, of

course, remove the ARCHIVE program cartrige as soon as ARCHIVE finishes loading, unless you want to use the ARCHIVE help facility.

If you do not have a formatted cartridge in drive 2, put one in now.

4 Press any key to continue
(If you don't do this ARCHIVE will start anyway within a few seconds)

ARCHIVE's main screen, divided into three areas, appears.

STARTING ARCHIVE FROM THE BASIC SCREEN

If you selected a screen type before inserting a cartridge, or if you have just finished using another program, you will be looking at the QL BASIC SCREEN.

To start ARCHIVE:

1 Insert your copy of the ARCHIVE cartridge in the left-hand Microdrive (drive 1)
Check that you have a blank, formatted cartridge in Microdrive 2.

`lrun mdv1_boot` ↵

2 Type `lrun mdv1_boot` and press **ENTER**
This asks the QL to look for a file called `boot` on the cartridge in Microdrive 1, then read it off the tape and start it running in the QL. This `boot` file in turn finds the ARCHIVE program file and starts to load it from the tape into the QL.

Note the space after `lrun` and the '_'.

The Microdrive will whirr and ARCHIVE's starting display will appear as described above.

The term to 'boot' is computer jargon for finding the first program and reading it into the computer so that the machine can load and use larger programs. The '`boot`' program is normally the very first program run in most computers. The SUPERBASIC command `lrun` is short for 'load and run'.

Display widths

A domestic television is designed to show TV pictures, but can also be used with the QL. A monitor has been specifically designed to work with a computer, and shows information much more clearly. Because of this greater clarity a monitor shows 80 characters across the screen. A domestic television can do so too, but it is easier on the eyes with 64 characters across. There is also a facility for a 40 character display width (see the `mode` command in chapter 10). This is unnecessary in the UK, but may be useful in other countries where the TV standards differ. ARCHIVE's display will vary slightly depending on the type of screen you are using.

The essence of ARCHIVE's display, however, is the same whichever screen you are using.

Note: The examples have been written assuming you have a 64 column display on a TV. The display width you have makes no difference to the way ARCHIVE works, but any screen layouts you design yourself might have to be changed.

THE SCREEN

At the moment you are in ARCHIVE's first and main screen. Look at the screen. If the definition of your picture isn't very good, try adjusting your television controls such as 'colour', 'contrast' and 'brightness'. Often the best settings for the QL are not the same as the best ones for watching television.

Notice that the screen is divided into three areas. These are the CONTROL AREA, the WORK AREA and the DISPLAY AREA.

Control area

The oblong box at the top of the screen is the CONTROL AREA. It's there to remind you what to do. At the moment you can:

- Obtain HELP by pressing **F1**
- Turn the CONTROL AREA and its prompts on or off by pressing **F2**
- Look through the display of commands by pressing **F3**
- Type in a command
- Press the **ESCAPE** key – (although this will have no effect just yet)

Work area

The area at the bottom of the screen is the WORK AREA. All commands are typed in here, and ARCHIVE displays error messages here if you make a mistake. You also write and edit program lines here.

Commands

Like all computer programs, ARCHIVE needs to be told what to do – you must give it commands. A command is a one word instruction you type in. It tells ARCHIVE to carry out further instructions to accomplish a task. ARCHIVE reports back for another command when it has finished. Note that, unlike QUILL, EASEL and ABACUS, commands in ARCHIVE must be typed in full.

ARCHIVE understands all the commands that appear in the list in the CONTROL AREA, and you can also create your own.

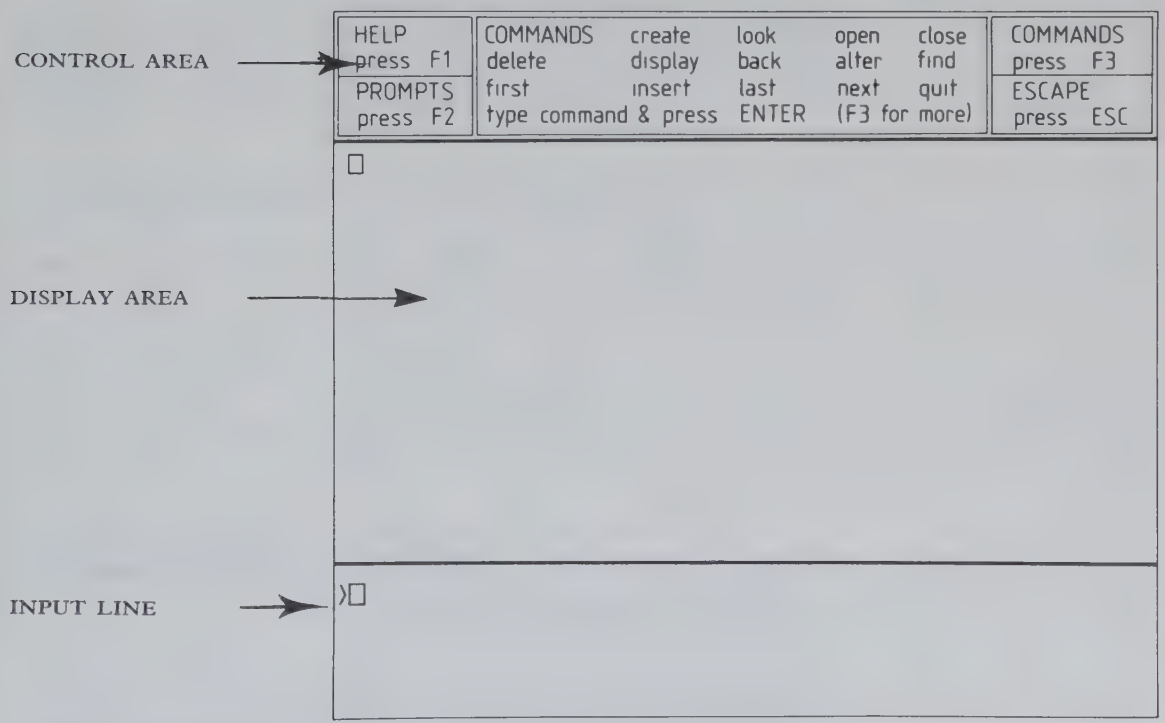
Some commands need further information (other words or numbers) before they can be performed correctly. The maximum length of a command line is 128 characters, including spaces between words.

The prompt

As you type in a command, the letters appear beside the > sign in the WORK AREA. The > symbol is called the prompt. It reminds you that ARCHIVE is waiting for you to type in a command. This line being typed is called the input, or command, line.

Display area

The middle of the screen is the DISPLAY AREA, in which the results of commands are displayed.



Notes on the Screen

CONTROL AREA
Contains reminders of options – tells you what you can do

WORK AREA
All commands and program lines are typed in here

DISPLAY AREA
Displays your information, and the results of commands

PROMPT
Shows that ARCHIVE is ready to receive a command

CURSOR
A red (or white in black and white TVs) rectangle indicating where any letters you type will appear

USING THE FUNCTION KEYS

The keys at the left of the keyboard are called the FUNCTION KEYS. They are labelled **F1**, **F2**, **F3**, **F4**, and **F5**. Here is a quick guide to what they do.

F1: The HELP key

All the HELP messages are stored on your copy of ARCHIVE, so ensure that this is in drive 1 (if you wish to use it).

F1 You can ask ARCHIVE for help at any time by pressing **F1**. Press it now. The Microdrive whirrs, the screen changes and a list of topics appears. The cursor sits at the end of the list, next to a question mark. ARCHIVE is asking you which general area of its capabilities you want to know about.

The list is probably meaningless to you right now. To find out more, select one of the topics.

You only need type enough letters to distinguish the topic you want from the rest.

To find out more about 'File Commands':

fi 1 Type `fi`, or `FI`, (either will do).
Two letters are needed to distinguish this option from FUNCTIONS

↵ 2 Press **ENTER**
The screen clears and information on commands associated with files appears. Additional help is available on individual commands. To find out more about `create`:

cr 3 Type `cr`

↵ 4 Press **ENTER**
The screen clears again and ARCHIVE displays information on `create`.

Other choices give you other lists. Make a foray into the HELP SCREENS if you like. Don't worry if you don't understand all that you see – it will make sense as you read through the book.

Return to the main screen in one of two ways:

either

ESC Press the **ESCAPE** key
This returns you immediately to the place you were when you originally requested help

or

↵ ↵ ↵ Press **ENTER**
This returns you to the previous HELP SCREEN which offered you a list of choices. Each time you press **ENTER**, ARCHIVE returns you to the previous HELP SCREEN, until you reach the main ARCHIVE screen at which you typed in **F1**.

You can press the HELP key at any time, without fear of losing your current work. The HELP SCREEN that appears will always relate to what you are doing. If, for example, you press HELP while you are using the `insert` command, the information that appears will be about that command. When you finish, press **ESCAPE** or **ENTER**, and ARCHIVE will take you back to the exact place you were before pressing the `help` key.

If the ARCHIVE cartridge is not in the left hand drive, ARCHIVE merely responds with the message '`No help file on mdv1 -- Press any key to continue`'. Press any key and you are immediately returned to whatever you were doing. You will not lose your work if this happens.

F2: Turning the CONTROL AREA on and off

As you become more familiar with ARCHIVE, you may not need to be reminded what to do next. You can temporarily erase the CONTROL AREA by pressing **F2**.

F2 Press it now
The CONTROL AREA vanishes, and the DISPLAY AREA now occupies the entire top part of the **F2** screen. Press **F2** again, and the CONTROL AREA reappears. You can use this key at any time, whatever you are doing.

F3: The COMMANDS List

Pressing the **F3** key displays more of ARCHIVE's commands in the central box in the CONTROL AREA.

F3 Press **F3**: a new set of commands appears.

F3 F3 Press **F3** twice more
Each time you press it, further commands are displayed in the box. Press it a fourth time and the first set of commands (beginning with `create`) reappears.

All the commands are available at any time. The four boxes are used for display purposes, so you do not need to switch boxes before using a command (unlike QUILL).

F4

This key has different functions depending on where you use it. These are discussed later in the relevant sections of the book.

F5

This key has three main functions, depending on where you are:

- saves information that you type in that needs to be stored on the Microdrive cartridge.
- recalls a line for editing when writing programs.
- calls back to the command line (after the > prompt) the last line typed in, so that it can be changed or used again.

EDITING TEXT IN ARCHIVE

Entering and editing lines

You should be looking at ARCHIVE's main screen, with the cursor sitting after the > prompt in the work area.

Try out the keys as used in the following examples.

`Fred` 1 Type in your name.
The letters appear after the > prompt. Each time you type a letter, the cursor moves to the right. The next character typed appears where the cursor is, and it moves right again.

[↵] 2 Press **ENTER**
This tells ARCHIVE you have finished the line, and are ready for ARCHIVE to interpret and act on it. However, because it is not a recognised command, ARCHIVE displays an error message. This is a message which ARCHIVE displays on the screen to tell you that it does not understand the command that you have typed in. There is a detailed description of these error messages in the section entitled 'Getting Out Of Trouble' in the Appendix.

[F5] 3 Press function key **F5**
Your name reappears. **F5** always recalls the last command or command line entered in the input line. This is very useful. If you type an error you can recall the line with the error, correct it (using the editing keys discussed below), and then carry on. Practice this now on the line with your name.

[↓] 4 Press the down arrow
The cursor jumps to the end of the line.

[↑] 5 Press the up arrow
The cursor jumps back to the beginning.

CAPS LOCK 6 Press **CAPS LOCK**
Any letters you type from now on will be in upper case until you press **CAPS LOCK** again. Only letters are affected.

NAME 7 Type in the word NAME

SPACE 8 Press the **SPACE BAR** to insert a space between the word NAME and your name, to make it easier to read. Notice that the capital letters appear at the start of the line, pushing the other letters along to the right.

CAPS LOCK 9 Press **CAPS LOCK** again

◆ 10 Press ◆ and hold it down
The cursor travels right over the letters and spaces underneath without affecting them. When it gets to the end of the line, it stops because it can only move through text. Lift your finger off the key. Holding a key down causes the QL to act as if you were pressing the key repeatedly.

SHIFT ◆ 11 Hold down the **SHIFT** key and press the left arrow. The cursor jumps to the beginning of the word.

Erasing text

CTRL ◆ 12 Hold down the **CONTROL** key and press the left arrow once.
If you are not at the beginning of the line, the character immediately to the cursor's left is erased, and the line closes up to fill the space.

CTRL ◆ 13 Hold down the **CONTROL** key and press the right arrow key once. This time, the character underneath the cursor is erased.

Inserting text

14 To restore the missing letters, type them in again. They appear under the cursor, and the character previously under the cursor moves to the right.

Moving within the text

SHIFT ◆ 15 To jump one word to the right, hold down the **SHIFT** key and press the right arrow.

CTRL ◆ 16 To delete the whole line to the left of the cursor, hold down **CONTROL** and press the up arrow. The whole line vanishes.

ESC 17 Press **ESC**
ARCHIVE displays <ESC> below the line and a new > prompt appears beneath.

No error message was displayed this time, because pressing **ESC** on an input line causes the line to be abandoned. Anything you have typed will be ignored completely.

KEYS USED IN EDITING – The Line Editor

CURSOR KEYS *Moving*

⬌	moves the cursor to the end of a line
⬌	moves the cursor to the start of a line
⬍	moves the cursor one character to the right
⬍	moves the cursor one character to the left
SHIFT ⬍	moves the cursor one word to the right
SHIFT ⬍	moves the cursor one word to the left

Erasing

CTRL ⬍	erases the character underneath the cursor
CTRL ⬍	erases the character to the left of the cursor
CTRL ⬌	erases all the text to the right of the cursor
CTRL ⬌	erases all the text to the left of the cursor

OTHER KEYS

↵	lets ARCHIVE know that a line is complete
ESC	tells ARCHIVE to abandon a line or command
F5	recalls the command line, or current line for editing
Holding down key	repeats character as long as key is held down

LEAVING ARCHIVE

The QUIT command

When you want to leave ARCHIVE, don't simply switch the QL off. The `quit` command shuts down ARCHIVE properly, without risk of losing any data that you may have typed in.

`quit` 1 Type `quit`

↵ 2 Press **ENTER**
ARCHIVE returns you to the screen you see when you first switch the QL on (which prompts you to press either F1 or F2) (ARCHIVE version 1), or to the QL BASIC SCREEN (ARCHIVE version 2).

3 Now remove the cartridges from the Microdrives and switch off, or use another program.

Note: If you use `quit` in Version 2 of ARCHIVE, you cannot then restart with the '`lrun`'. You must take out the Microdrive cartridges, press **RESET**, and start again.

Return to this chapter for reference whenever you feel the need. The significance of all that has been said in this chapter should become clear as you progress through the book.

The First File

ARCHIVE'S primary purpose is to store and retrieve information. This information is stored on tape in files (where similar information is grouped together). By the end of the chapter you will have learnt, in a simple way, both how to store information in a file, and to retrieve it.

The following features of ARCHIVE are introduced:

- The ARCHIVE file-handling commands:
  `create` – create a new file structure
  `dir` – list the files on a tape
  `close` – close and save a file

- The ARCHIVE record-handling commands:
  `display` – display a file
  `insert` – insert information into a file
  `alter` – alter information in a file
  `next` – move forward through a file
  `back` – move backwards through a file
  `first` – move to the first record in a file
  `last` – move to the last record in a file
  `find` – find information
  `continue` – find more information

Introduction

This chapter explains ARCHIVE's primary function: to store and retrieve information. You will use ARCHIVE to create and use a simple database file. The finer points of using these commands will be covered at greater length in the next chapter.

As you follow the example, you will encounter various prompts in the `control area` which you may not understand – these too will be more fully explained later. It is not important to understand them at this stage.

Don't worry about making mistakes. If you do something ARCHIVE doesn't like, an error message will appear. All you have to do is type the words again correctly.

Example: Miranda organises her wedding

Whenever Miranda isn't gazing starry-eyed at Nigel, she's using ARCHIVE to help organise the Big Day. She keeps her wedding guest list on the computer, and plans to use ARCHIVE to address the wedding invitations, log the replies, and keep track of both numbers coming and who gives what. She toys with the idea of letting ARCHIVE work out the seating arrangements for the wedding lunch; it couldn't do worse than her mother, and would certainly make less fuss.

She keeps thinking of things to do, before and after the wedding, or remembering things she should already have done. Looking at all her scraps of paper, and the lists she's rewritten umpteen times, she decides to computerise her notes as well as her address book.

CREATING A FILE

Miranda's first task is to use ARCHIVE to create a file, which she calls `memos`.

This is similar to putting a label on a cardboard file, which is then filled with paper records and stored in a drawer or filing cabinet. A file keeps information together, and separate from other files. The file's name enables the information to be found quickly.

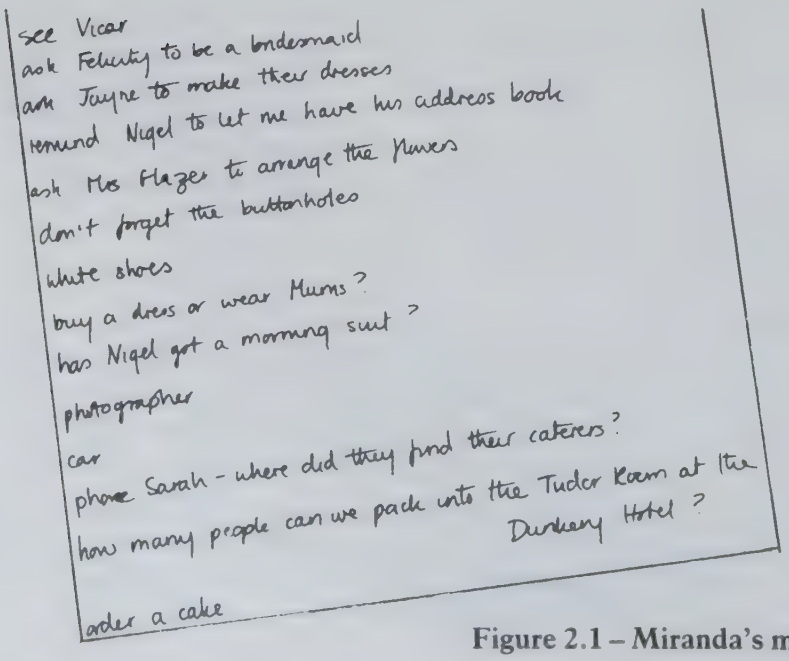


Figure 2.1 – Miranda's memos

Instructions

1 Make sure that you have the ARCHIVE program loaded (see last chapter), and that you have a blank, formatted tape in drive 2.

`create` ↵ 2 Type `create` and press **ENTER**
ARCHIVE provides two quotation marks after `create`.

`memos` 3 Type `memos`
This is the new file name. Everything you type appears between the quotation marks. If you make a typing mistake, erase the characters you have typed using any of the line editor features (see the end of Chapter 1).

When any mistakes are corrected:

↵ 4 Press **ENTER**
The cursor moves to the next line, without a > prompt, and the CONTROL AREA at the top of the screen changes. It lets you know what your options are at this point.

ARCHIVE is waiting for additional information which it needs to create a new data file: a name or label for the information you want to store. In this case, we want to store reminders, or notes.

`note$` 5 Type `note$`
This is the name you will store the information under. Notice that the word ends with a dollar sign, '**$**' (**SHIFT** and 4). This tells ARCHIVE that text of the note (ie letters and numbers) will be stored under the name '`note$`'. If you leave the dollar sign off, ARCHIVE will only store numbers.

↵ 6 Press **ENTER**
The cursor moves down a line. The CONTROL AREA does not change. ARCHIVE is waiting either for another name, or to be told that you have finished creating the file. No more information is going to be stored in the memos file, so you have finished. If you make a mistake, refer to the section below called 'Coping with mistakes in `create`'.

↵ 7 Press **ENTER** again
This ends the `create` command. ARCHIVE assumes that you are finished if you press **ENTER** on a blank line. The word `endcreate` appears automatically, and the Microdrives begin to whirr. ARCHIVE is saving the details of the file onto tape.

Note: You could have typed in `endcreate` yourself, in full, but it is easier to let ARCHIVE do it for you. Either way, `create` must end with `endcreate`, or the file will not be created.

The > prompt reappears, ready for another command. Once again the CONTROL AREA displays a list of the commands.

ARCHIVE does not tell you that it has successfully created the file. The only indication is the absence of an error message. ARCHIVE makes no assumptions about what you want to do with the file once you have created it.

The file is now 'open' and ready to receive the information you want to store in it.

Coping with mistakes in CREATE

Mistakes on a line:
If you notice a mistake in a line as you are typing it, before you press ENTER, the line can be edited using the normal keys (see the end of Chapter 1).

Mistakes on a previous line:
It is not possible to go back a line in `create`. Pressing F5 in `create` to recall a line either will not work, or will have some other effect. You must abandon the `create` command and start again.

To escape:
To abandon `create` completely, before you get to `endcreate`, press ESC. ARCHIVE displays `<ESC>` and the > prompt reappears. Anything you have typed in `create` is ignored, and the file will not be created or saved onto tape. To start again, type `create` after the > prompt. Your file structure is only saved if `endcreate` is entered.

Error messages:
ARCHIVE needs to be told what to do very precisely. If ARCHIVE does not understand what you type in, an error message appears in the WORK AREA, the last command you gave is abandoned, and the > prompt reappears. In this case, anything you may have typed in in `create` is simply ignored, and has no effect whatsoever. To start again, type in `create` after the > prompt.

INSERTING INFORMATION INTO THE FILE

Having created her file, Miranda is ready to add information to it.

`insert` ↵ 1 Type `insert` and press ENTER
The CONTROL AREA changes to display some of the keys ARCHIVE expects you to use. Ignore this for the time being.

The cursor in the WORK AREA has tucked itself out of the way on top of the > prompt. A second cursor appears in the DISPLAY AREA. During `insert` this is where characters will appear as you type. Until the > prompt reappears, ARCHIVE expects you to insert records into the file (ARCHIVE calls the items that make up a file 'records' – in this case, each individual note is a record).

The top line of the DISPLAY AREA now reads:

```
Logical name : main
```

This is a temporary name which ARCHIVE has allocated to the file. It has nothing to do with logic, and its purpose is explained further on. Ignore it for now.

The screen contains the following information:

```
DISPLAY AREA      :   Logical name: main
                      note$      : []

WORK AREA         :   >create "memos"
                  :   note$
                  :   endcreate
                  :   >insert
```

ARCHIVE is waiting for you to type in the first note.

`see Vicar`

4 Type `see Vicar`
As you type, the letters appear after `note$ :` in the DISPLAY AREA.

F5

5 Press F5 (or **ENTER** in version 2)
to save the record and to move on to the next one. (Pressing **ENTER** here merely returns the cursor to the start of the line in version 1.) The words you have just typed vanish. The cursor returns to the colon. The record containing the note `see Vicar` has been inserted into the file `memos`. ARCHIVE does not tell you that it has added the record to the file.

ARCHIVE is waiting for you to insert another record. All you can do in the command `insert` is insert new records into a file. You cannot look at any other records.

`ask Felicity to be a bridesmaid`

6 Type `ask Felicity to be a bridesmaid`
If you make a mistake, edit the line using the normal keys.

F5

7 When the line is correct press F5.

The line you have just typed vanishes and the cursor returns to the colon.

`ask Jayne to make their dresses` F5

8 Type `ask Jayne to make their dresses` and press F5.

The Function and Enter Keys in INSERT

Versions 1 and 2 of ARCHIVE use different keys in `insert`.

To Save a Record

Version 1	F5
Version 2	F5 or ENTER on the last field

Pressing ENTER on the last field in version 1 simply takes you to the first field.

To Leave INSERT

Version 1	ESC
Version 2	F4

F4 is a better way to leave `insert` than ESC, which implies an error.

The `memos` file now holds three notes:

```
see Vicar
ask Felicity to be a bridesmaid
ask Jayne to make their dresses
```

Leaving INSERT

A client squabbling with his neighbour's intrusive rose bush tears Miranda away from her engrossing task. But before rushing out to defend the aggrieved party, she wants to check through what she's written. She must first tell ARCHIVE that she has finished with `insert`.

ESC

1 Press ESC. (In version 2, press F4)
The DISPLAY AREA now shows an empty record. The > prompt and cursor reappear in the WORK AREA. ARCHIVE is waiting for another command.

Note: ESC abandons `insert` in ARCHIVE versions 1 and 2. In version 2, `insert` can be abandoned by pressing F4. This is a better key to press, as ESC causes an 'error' in ARCHIVE which collapses programs you write yourself (discussed in Chapter 9).

Displaying information

Miranda uses the `display` command to examine her handiwork.

`display` ↵

1 Type `display` and press ENTER
The last record you entered is shown. It's the current (most recently used) record: the one always shown by `display`.

MOVING AROUND THE FILE

The BACK command

To see the other items that you entered:

back ↵ 1 Type back and press ENTER
ARCHIVE immediately displays the previous record, which now becomes the current record. The > prompt reappears on the next line in the WORK AREA.

To move back again without retyping back:

F5 2 Press F5
The word back appears after the > prompt. F5 recalls the last command line entered.

To execute the command:

↵ 3 Press ENTER
ARCHIVE moves back one record through the file and displays it in the DISPLAY AREA.

F5 4 Press F5 again
The word back appears on the input line.

↵ 5 Press ENTER
This time the display may flicker, but it does not change. You are looking at the first record in the file. There is no record before the current record. ARCHIVE does not tell you this, but merely redisplays the first record.

The LAST, NEXT and FIRST commands

To go to the last record in the present file:

last ↵ 1 Type last and press ENTER
The DISPLAY AREA now shows the last record you entered.

next ↵ 2 Type next and press ENTER
The DISPLAY AREA does not change. There are no records after the last record. Once again ARCHIVE does not inform you that the end of the file has been reached.

first ↵ 3 Type first and press ENTER
You are again looking at the first record in the file.

next ↵ 4 Type next and press ENTER
The DISPLAY AREA now shows the next record in the file. In ARCHIVE's terms, it has become the current record.

The Current Record

The current record is the record which ARCHIVE is dealing with at present – this is not necessarily the record that is displayed on screen. It's like the page at which a book is open. When the book is first opened, the first page is the current page. As each page is turned, the latest page becomes the current page. In ARCHIVE the current record is always the last record moved to, looked at, added or changed. Some commands act on many records in sequence; the current record will be the last one in the sequence when the command finishes, (although there are some exceptions to this).

Commands which affect records affect the current record unless you specify otherwise.

ADDING NOTES TO THE MEMOS FILE

Miranda wants to enter some more records into the file.

Note: ARCHIVE should allow a maximum of 160 characters, including spaces, to be entered in any one line of a file when using `insert`. However, typing in that many characters can be impractical. You may find you have problems typing in the note about the Dunkery Hotel. If you have version 1, the display starts wrapping over onto the next line when it reaches the edge of the screen. When the next record is displayed, the wrapped over part of the previous record may remain on the screen. In some versions of ARCHIVE you are prevented from completing the line. Keep lines as short as is convenient.

It is possible to store up to 255 characters in one line in an ARCHIVE data file, but not by using `insert`. Other methods are described later.

`insert` ↵ 1 Type `insert` and press **ENTER**

```
remind Nigel to let me have his address book  F5
```
2 Type in Miranda's next note (above).

```
ask Mrs. Glazer to arrange the flowers  F5
don't forget buttonholes  F5
white shoes  F5
buy a dress or wear Mum's ?  F5
has Nigel got a morning suit ?  F5
photographer  F5
car  F5
phone Sarah - where did they find their caterers ?  F5
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?  F5
order a cake  F5
```

3 Type in the rest of Miranda's notes (above), pressing F5 when you've finished each one to save it.

When you have finished all of them:

ESC 4 Press ESC to leave `insert` (or press F4 if you have version 2 of Archive).

Note: in version 1, if you press ENTER instead of F5, that record is lost.

The `memos` file now contains the following notes:

```
note$
see Vicar
ask Felicity to be a bridesmaid
ask Jayne to make their dresses
remind Nigel to let me have his address book
ask Mrs. Glazer to arrange the flowers
don't forget buttonholes
white shoes
buy a dress or wear Mum's ?
has Nigel got a morning suit ?
photographer
car
phone Sarah - where did they find their caterers ?
how many people can we pack into the Tudor Room at the Dunkery Hotel ?
order a cake
```

FINDING AND ALTERING RECORDS

After struggling into Joyce's ruffled confection, Miranda decides to put in a hasty visit to *Pronuptia* instead. To change the note about the wedding dress,

`find "dress"` ↵

1 Type `find "dress"` and press ENTER
The screen shows the note `ask Jayne to make their dresses`.

The command `find` goes to the beginning of the file and examines each record in turn. ARCHIVE tries to find the letters inside the quotation marks in any text in the record and stops at the first record it finds containing these letters.

`continue` ↵ 2 Type `continue` and press ENTER
The screen shows the next record containing the letters 'dress':
`remind Nigel to let me have his address book`

The command `continue` will continue the last `find` command, looking through the remaining unsearched records. If it fails to find any further records which match, the last record matched remains the current record (just as `find` leaves the starting record as the current record if nothing is found).

F5 ↵ 3 Press F5 followed by ENTER
ARCHIVE continues its search and finds the record Miranda wants – `Buy a dress or wear Mum's?`. In this case, she would have done better to look for the characters '`dress`' or '`Dress`'. The space before the `d` helps narrow the search to the word you want. The command `find` will find the matching letters whether they are in upper or lower case.

Note: When using the `find` command, type in only as many letters as will distinguish the record from any others. You do not need to type in a whole word or phrase.

Altering records

Always a stickler for detail, Miranda wants to change the information in her record:

`alter` ↵ 1 Type `alter` and press ENTER
Notice that the prompts are the same as for `insert`. The cursor in the `work area` appears on top of the > prompt and another cursor appears in the `display area` at the beginning of the record.

↓ 2 Press the down arrow to move to the end of the line.

CTRL ← *(15 times)* 3 Press CTRL at the same time as the left arrow, and delete the words `or wear Mum's ?` one character at a time.

F5 4 Press F5
The > prompt reappears in the WORK AREA, waiting for another command. Unlike `insert`, you don't have to press ESC to return to the commands prompt >. ARCHIVE automatically returns you to commands after altering one record.

Note: If you decide (before pressing F5) you do not like your alteration, use F4 in version 2, and ESC in version 1, to abandon `alter`.

Finding and altering more records

Imagining herself a vision in pink and cream taffeta, Miranda suddenly remembers the name of the local photographer.

`find` ↵ 1 Type `find` and press ENTER
Two quotation marks have appeared after the command and the cursor is waiting for you to insert the word you want to find. The CONTROL AREA tells you how to use `find` and prompts for the word or letters (including spaces, if any) to be found.

Note: Some commands, like `create` and `find`, need to be followed by letters between quotation marks. If you press **ENTER** after typing in only the command, ARCHIVE automatically supplies the quotation marks, which means you don't have to type them in yourself.

`phot` 2 Type `phot`
The letters '`phot`' appear between the quotation marks.

↵ 3 Press **ENTER**
The record with the note '`photographer`' appears in the display area.

`alter` ↵ 4 Type `alter` and press **ENTER**
Once again the cursor in the display area appears at the beginning of the record.

`b` 5 Type `b`
As soon as you type the first letter, the rest of the line vanishes. ARCHIVE always erases a line when you type a letter in the first position in `alter`.

Note: If you want to insert characters at the beginning of a line in `alter`, first press the up arrow followed by the letters you want to insert.

`ook Pickwick the photographers` F5
6 Type `ook Pickwick the photographers` and press **F5**

Note: To abandon `alter` without saving any changes, press **ESC**. The altered version of the record will remain on the screen, but has not been saved. The original, unchanged record has been left intact. To check this, type the commands `back` and `next`, this will bring up the unaltered record in the CONTROL AREA.

The `memos` file now contains the following notes:

```
note$
see Vicar
ask Felicity to be a bridesmaid
ask Jayne to make their dresses
remind Nigel to let me have his address book
ask Mrs. Glazer to arrange the flowers
don't forget buttonholes
white shoes
buy a dress
has Nigel got a morning suit ?
book Pickwick the photographers
car
phone Sarah - where did they find their caterers ?
how many people can we pack into the Tudor Room at the Dunkery Hotel ?
order a cake
```

SAVING THE FILE

Gazing proudly at her handiwork, Miranda wants to save what she has done, lest anything untoward like a power failure destroy the fruits of her labour. First she checks the files that are on her Microdrive. Then she can save her data by closing the files that are open.

Memory

The QL has two kinds of storage for data as well as programs: temporary memory and long-term storage.

Temporary memory lasts as long as the power remains on. The QL and ARCHIVE do all their work in the temporary memory, since it stores and retrieves information many, many times faster than the QL's long-term storage – the microdrive.

Long-term storage on the QL takes the form of tape in a microdrive cartridge. Files are stored on this in exactly the same way that songs might be recorded on an ordinary audio cassette. They can of course be erased or rerecorded. They can also become unusable through a variety of mishaps, ranging from physical damage to exposure to magnetic fields.

Temporary memory

Temporary memory is more usually referred to in computer jargon as RAM, which stands for *Random Access Memory*. Information is stored as electronic impulses in microchip circuits, and can be retrieved (or 'accessed') in any order, ie. randomly. A tape on the other hand, has to pass over information sequentially to find any one piece of it. Also, being mechanical rather than electronic, tapes do this much, much more slowly than the electronic signals in RAM.

There is a limited amount of temporary memory, and everything ARCHIVE does uses up some of it. ARCHIVE itself occupies a large amount, and each file you use takes up more.

Each time you switch off the QL, or leave ARCHIVE, everything in the temporary memory is lost. If you wish to keep this information, it must first be recorded onto tape.

Other forms of long term storage include discs (which are faster than tapes) and ROM chips, which are as fast as RAM, but contain only fixed information like programs.

Listing the files on tape – the DIR command

`dir` ↵

1 Type `dir` and press **ENTER**
The CONTROL AREA changes and prompts you to enter a drive name. Quotation marks appear after the word `dir`. ARCHIVE is waiting to be told which microdrive you want to see a directory for.

↵ 2 Press **ENTER** again
If you do not tell ARCHIVE which drive to look at, `dir` will look for your files on the right hand drive (drive 2). If you don't want ARCHIVE to make assumptions for you, type in a drive name between the quotation marks, eg. `mdv1_`.

The DISPLAY AREA now shows a directory of microdrive 2. This is a list of the files stored on the tape in the drive. It also shows how much space is left on the tape. Miranda's file appears as `memos_dbf`. The letters on the end stand for Data Base File. See the box for more information on file names.

Note: `dir` shows how much space there is left on the tape in terms of sectors. When you format a tape (see Chapter 1), the tape is divided into about 218 sectors. The message `13/218 sectors` indicates that there are 13 sectors left unused on the tape.

Closing files

When the file `memos` was created (in temporary memory), the empty structure containing the data name (`note$`) was saved onto tape, but with no data as you had not typed any in.

Since then, several records have been added to the file in the temporary memory. Some, but not all, may have been stored on tape as the area in temporary memory set aside for `memos` began to fill. The copy of the file on the tape is incomplete.

ARCHIVE can only look at, alter or add to the contents of a file if it is in the temporary memory. The file is said to be *open*.

Any changes or additions made to the file while it is open take place only in the temporary memory. To save these changes, the data in memory has to update/replace the data in the file held on tape. This is done properly by *closing* the file.

When a file is closed, the information about the file in *temporary memory* (which may include any records that were changed or added) will be wiped out. The file therefore can no longer be looked at or altered until it is reopened. Files are re-opened by copying up to 500 characters of the file held on tape into the temporary memory. This does not affect the file held on tape in any way. It remains exactly where it is, and contains exactly the same information as the newly opened file in the temporary memory.

Closing a data file in ARCHIVE is like saving a file in SUPERBASIC, ABACUS, EASEL and QUILL.

Note:

- The command `create` opened the file `memos` automatically. `endcreate` saved the empty file onto tape. If a file called `memos_dbf` already existed on the tape, it would have been replaced without warning by the empty file.

- Be careful when creating files to make sure that the file name you choose does not already exist, or the existing file could be lost by being overwritten with the new file.

`close` ↵ 1 Type `close` and press **ENTER**
The microdrives whirr, and all the records in the file are recorded onto the tape. The file on tape is now being replaced by the copy of the file with the newly added records. Just as `endcreate` gives ARCHIVE the instruction to save the file structure, so `close` gives the instruction to save the records, the body of the file, onto tape.

The > prompt reappears after a few seconds. ARCHIVE is ready to receive another command even if the drives are still working.

Note: Save your work by closing files regularly. Once opened, all additions and changes to a file take place initially only in the QL's memory. Some records may be periodically saved on tape when the temporary memory requires it.

If the power fails or the machine is switched off for any reason, everything in memory is lost. Only information stored on tape will be available for use when the QL is switched on again. Files which are not closed properly by `close` or `quit` may not be usable.

It is surprising how frustrating it can be to lose even a few minutes' work.

Drive names

Any file name may be preceded by a drive name, to let ARCHIVE know which drive to look on to find the file. If no drive name is given, ARCHIVE looks on the right hand drive, microdrive 2. A file on microdrive 2 might be specified as `mdv2_memos_dbf`, where `mdv2_` is the drive name, `memos` is the file name and `_dbf` is the file extension.

Note that a drive name is incomplete without an underline after it, even when used on its own. eg. `dir "mdv1_"`

File Names

A full file name may consist of three parts, each joined by '_', the underline character. The three parts (in the correct sequence) are:

(drive name)_(file name)_(file extension)

File names:

- are used to identify files stored on tape
- may be up to 8 characters long
- must begin with a letter
- must have no spaces

File name extensions:

- may be up to three characters long (letters or numbers)
- are joined to the file name by an underline, '_'
- are used to identify types or groups of files held on tape

  'memos_dbf' is a Data Base File, as distinguished from a file containing commands, which might end with _prg.

  Usually there is no need to type in the extension – ARCHIVE supplies it automatically. Alternatively, you can provide your own. You could, for example, tag the extension _old (in place of _dbf) onto out of date copies of data files you still want to keep. Any extension you give overrides ARCHIVE's assumptions, but you must remember to use it every time you make reference to the file.

  The following ARCHIVE commands create or expect these extensions:

```
_dbf : create,open,look,import
_prg : save,load,merge,run
_scn : ssave,sload
_lis : spoolon
_exp : import, export
```

SUMMARY

Miranda has made a new ARCHIVE file called memos using the create and endcreate commands. Within this file she makes a heading, note$ and she uses the insert command to put a series of important items under this heading. The dollar sign ,$, after note indicates that the items or records are all to be regarded as text by ARCHIVE.

Miranda uses the `back`, `first`, `last` and `next` commands to move around inside her file so that she can view each record in turn. She also uses the `alter` and `find` commands to find a certain place in the file and change the record. The `close` command is used to completely transfer the ARCHIVE file from the QL's temporary memory to its long-term storage.

Helpful hints

- Use upper and lower case for all your data (except special codes). It is much easier to read than capital letters.
- Keep your lines of data to less than one screen width in length. ARCHIVE can store more characters, but they are difficult to amend and display. There is rarely any need to store really long lines. Most data is stored for printing at some stage, and ordinary paper is only about 80 characters wide.
- Press F5 to repeat commands. Use the editing keys to change the command if it was wrong and caused an error.

Multiple Files

In this chapter, you will learn how to create more complex files and to use more than one file at once.

The following ARCHIVE commands are featured:

- These ones are examined in detail:

`create`	– create a new file structure
`display`	– display current file
`insert`	– insert records
`alter`	– alter records
`close`	– close a data file

- These ones are introduced:

`open`	– load a file into temporary memory
`look`	– load a file into temporary memory to look at only
`new`	– clear temporary memory
`delete`	– delete current record
`backup`	– copy a file
`format`	– format a cartridge
`sprint`	– show the current record in the `displayed` file
`append`	– insert a record, using current field values

Points to remember

- `create` takes effect until the word `endcreate` is entered.
- `insert` goes on inserting records until you press ESC (or F4 if you have version 2).
- `alter` affects only the current record.
- Insertions and amendments do not take effect until F5 is pressed.
- `close` data files to save any changes or new records onto tape.

- Pressing ESC causes ARCHIVE to stop or ignore what you were doing.

THE SECOND FILE

Example: The guest list

Miranda now casts a beady eye on the tentative guest list which she and Nigel, with a little help from Joyce, have drawn up on paper.

First she writes down on paper the information she wants to store about her guests. She needs the addresses first to send the invitations, and later to send thank you letters for her presents. She is also going to use the list to keep track of the replies, the number of people who will come and those who will need to stay overnight.

This file is more complex than the `memos` file, where only one item of information was stored in each record in the file. Here, each name and address and associated information constitututes a record. The individual items of information within a record are called *fields*.

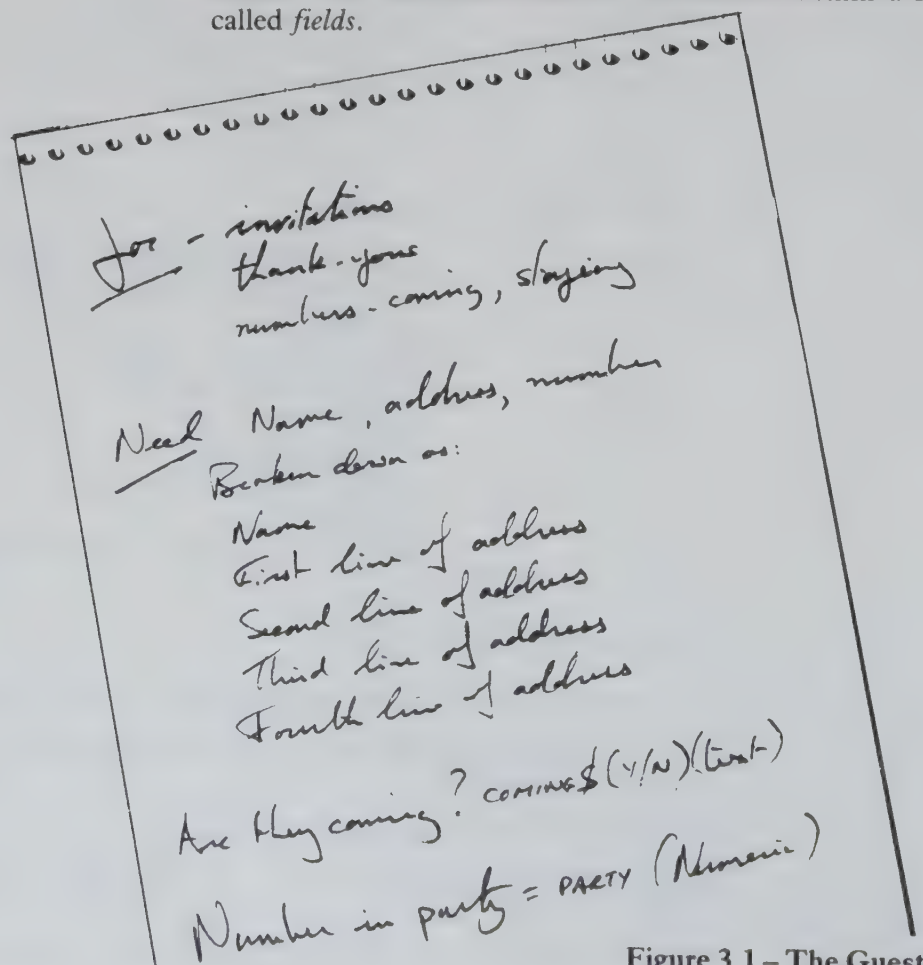


Figure 3.1 – The Guest File

Files, records, fields

Files a file is a collection of records on a given subject. It is stored on tape with a file name so that it can be recalled. For example, a cook might create a file called 'Recipes', a librarian one called 'Books', and a doctor one called 'Patients'.

Records the component parts of a file, such as one recipe, the details about a single book, or a patient's record card; these have similar layouts, but contain different information.

Fields the component parts of a record. For example, the ingredients and instructions in a recipe, the author and title of a book, the name and address of a patient.

Limits you may create as many files as you wish, if you have enough tapes to store them on. A text field may be up to 255 characters long (but only 160 characters are affected using `insert` and `alter`. A record may have up to 255 fields (although `alter` and `insert` affect only as many fields as there are lines on the screen).

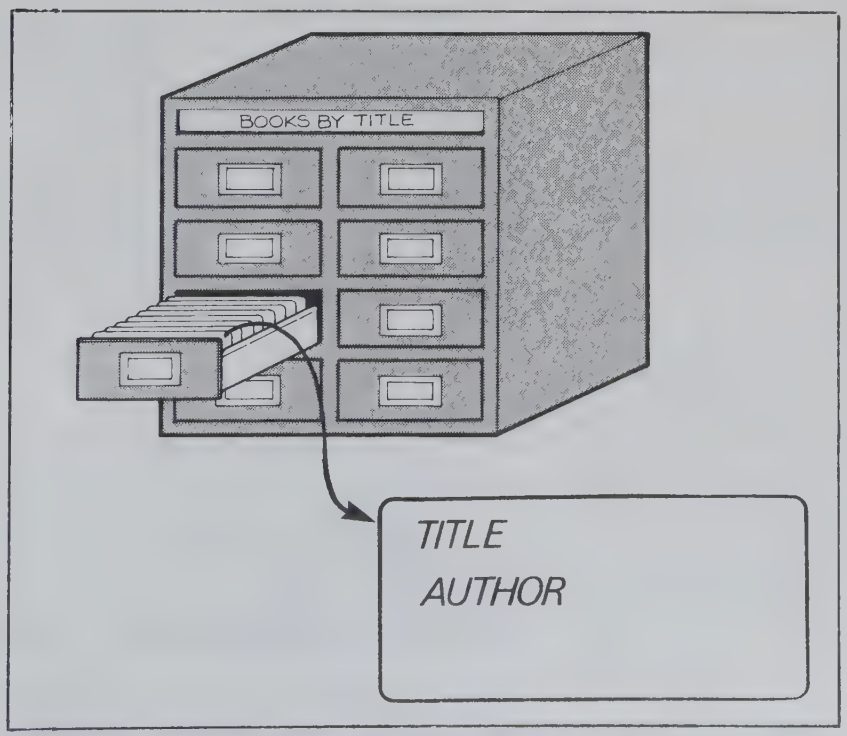


Figure 3.2 – Files, records, fields

Getting started

If you do not have ARCHIVE loaded, load it now, as described in Chapter 1. In drive 2 you should have the data cartridge with the `memos` file that was created in Chapter 2 on it. If you do not know what is on your drive 2 cartridge, type `dir` and press **ENTER** twice to see a list of the files on drive 2. Change the cartridge if it is the wrong one. Insert another, and repeat the `dir` command and so on until you find `memos_dbf`.

If you have any data files open, `close` them.

THE GUESTS

Miranda decides to call this file `guests`.

Creating the file

`create` ↵

1 Type `create` and press **ENTER**
Notice that the CONTROL AREA at the top of the screen changes. The list of commands has been replaced by some information about the command `create`. You are being prompted: `ENTER THE FILE NAME.`

↵

2 Press **ENTER** again
ARCHIVE prints an error message `bad file name`, and the prompt reappears. Don't worry if you don't always understand error messages. They are often not very helpful. They are explained in full in the appendix.

Note: The command `create` must be followed by an acceptable file name. ARCHIVE attempted to interpret the whole command, but failed to find a name at all. Errors always cause the last command entered to be abandoned, and the > prompt reappears. When this happens, start again.

`create` ↵

1 Type `create` and press **ENTER**

`guests`

2 Type `guests`
The word appears between the quotation marks.

↵

3 Press **ENTER**
The cursor moves to the next line. Your command was accepted this time.

From now until the prompt reappears ARCHIVE expects one of two things each time a new line starts – a field name, or an `endcreate`.

The `memos` file had only one field, `note$`: this file has many.

If you make a mistake and press **ENTER** after typing in a line, it is not possible to go back and correct it.

If it is a mistake that ARCHIVE will not accept, such as an over-long field name, ARCHIVE will print an error message and abandon `create`. You will have to start all over again by typing `create`.

If you end `create` before you mean to, by pressing **ENTER** twice, it is not possible to call back what you have done and carry on adding fields. Type `close` and start again.

Be very careful when entering field names in `create`. Always write them down on paper before you start.

Field names and types

Field Names

- may be up to 13 characters long
- must begin with a letter
- must have no spaces
- cannot be the same as commands or other key words which ARCHIVE has reserved for its own use

There are two kinds of field:

Number fields

- may only contain numbers. They are also called numeric fields.
- can be added, multiplied, divided or have other calculations performed on them.

Character fields

- may contain text in any combination of letters or numbers. They are also called 'strings', as they consist of a string of characters.
- String field names always end in a dollar sign, `$`.
- Numbers held in strings may be used in calculations, but only if they are first converted to numeric format (using the `str()` function described in Chapter 9).

Note: If you enter a field name with a space in it, ARCHIVE will not print an error message. However, the field name will be only the first word. Since this is unlikely to end in a `$` sign, the field will be numeric whether this is what you wanted or not. eg. `First name$` becomes a field called `First`.

Now enter the field names for the information about the guests.

`name$` ↵ 1 Type `name$` and press **ENTER**

Errors in CREATE

Changing a field name while typing it:
Before you press ENTER, you can use the line editing keys to change mistakes.

Changing a field name on a previous line:
You cannot go back up a line to correct it. If it is important, you will have to abandon `create` (press ESC) and start again. Everything you have typed in `create` will be ignored and lost.

Ending CREATE too soon:
If you accidentally press ENTER on a blank line, and there is at least one field, `endcreate` appears, the file is created, saved onto tape and left open. You cannot call back and add to what you have done. Nor can you simply start again exactly as before. ARCHIVE does not let you open another file in temporary memory with the same logical name. ARCHIVE will print the error message `duplicate name` if you start again with `create`.

If this happens, type in `close` and press ENTER. This will close the file you have just created. To delete this incorrect file that is now saved on tape, type `kill "guests_dbf"` (`kill` is explained in Chapter 5). Then you will be able to start `create` again, opening a new file with the same name in temporary memory.

Note: ARCHIVE (version 1) does not check the tape to see if the file you are creating already exists, and will not warn you as it wipes out an existing file with the same name.

Errors:
If you enter anything that ARCHIVE does not understand, an error message appears. Errors cause whatever you are doing to be abandoned, and the > prompt reappears. Start again. Error messages are explained in the appendix.

```
address1$↵address2$↵address3$↵address4$↵coming$↵partyof↵
staying$↵
```

2 Type in this list of field names, pressing ENTER after each word. Note that `partyof` does not end in a dollar sign:

```
address1$
address2$
address3$
address4$
coming$
partyof
```

After typing in a couple of names, the cursor will be on the bottom of the screen, and the top line of the WORK AREA will disappear each time **ENTER** is pressed. This shifting of lines up the screen is called scrolling (like a scrolled document being rolled up as it is read).

The scrolling up and disappearance of previously typed lines is one reason why you should keep a written note of the structure of your work, to keep track of progress.

When you have typed in all the fields:

↵ 3 Press **ENTER** again
`endcreate` appears. The Microdrive whirrs. ARCHIVE saves the empty file on the tape.

The file is now open in the temporary memory, and its structure may be displayed or records inserted.

The > prompt reappears, ready for another command.

Displaying the structure

Miranda knows that if she has made a mistake in her file structure (the field names and types), she can later copy any data in the wrongly structured file into a new structure (*see Chapter 5*). This is not easy though, so she checks that she has created the structure correctly.

`display` ↵ 1 Type `display` and press **ENTER**
The DISPLAY AREA now shows the record structure for the file. No records are displayed because no records have been entered yet.

Notice that there is a zero after the colon opposite `partyof`. ARCHIVE puts a zero in empty number fields, and nothing at all in empty text fields.

Note: A zero is represented on the QL's display by an '0' with a '/' through it. This is to distinguish it more clearly from the letter 'O'.

Looking at the directory

To reassure herself that her file really has been saved onto tape, Miranda uses the `dir` command.

`dir` ↵ ↵ 1 Type `dir` and press **ENTER** twice
The DISPLAY AREA clears and the directory of Microdrive 2 now shows two files: `guests_dbf` and `memos_dbf`.

Changing the current file: Logical names

Thinking about the guest list reminds Miranda of some other things to note down before she types in the names and addresses. She

wants to make a present list, and to start thinking about arranging hotel accommodation for guests who need to stay overnight. Before she can enter these notes, she has to open the `memos` file.

Logical names

Logical names are needed to distinguish different files when more than one file is open at once. Commands which affect open files, like `close` or `next`, only act upon one file at a time. They can be used with the logical name to specify which file as in this example:

`close "main"`	closes whichever file has the logical name `main`
`next "main"`	moves on one record in the file with the logical name `main`.

A logical name is a temporary name which lasts only as long as the file is open. It in no way affects the file name on the 'physical' tape. It need not even be similar. It need not be the same the next time you open a file. The only requirement is that no two logical names are the same at once.

Logical names can be as short as one letter, which saves having to type the name in full every time. The file name on the tape usually needs to be as long as possible to be meaningful and remind you what the file contains.

ARCHIVE automatically allocates the logical name `main` to a file when you open it if you do not specify a logical name, and if no file which is already open has the logical name `main`. Avoid the logical file name `main`, and always open files with a logical file name of your own choosing.

Logical names may be up to 13 letters long; they must begin with a letter and have no spaces.

Logical names are useful shorthand for designating which file you are working with. Their uses will become clearer as you go on. Keep a note of which logical names you are using for each file.

`open "memos"` ↵

1 Type `open "memos"` and press **ENTER**
The error message `duplicate name` appears.

The reason for this is slightly complicated: when a file is created or opened, it must be given a temporary name by which ARCHIVE refers to it while it is open. Each open file must have a different temporary name. The *temporary name* need *not* be the same as the file's *filename* used to store it on tape.

This temporary name is called a *logical name*. It has nothing to do with logic. It is a programming term to differentiate the file, while it is open in temporary memory, from any other files that are open, and therefore in temporary memory, at the same time.

If you do not specify a logical name when you `open` or `create` a file, ARCHIVE supplies one; but ARCHIVE only has one name in its repertoire – `"main"`.

When `guests` was `created`, no temporary name was specified, so ARCHIVE automatically gave it the logical name `main`. Two open files cannot have the same logical name, so when ARCHIVE tried to give the file `memos` the name `main` too, the error message `"duplicate name"` appeared. You must give this second file a different logical name yourself. To do this, and get round the 'error':

```
open "memos" logical "m" ↵
```

2 Type `open "memos" logical "m"` and press **ENTER**
The > prompt reappears, indicating that the command was successful. The file `memos` is now open. The display area, however, has not changed.

USING MULTIPLE FILES

Displaying the current file

Miranda now has two files open, `guests` which is held as logical name `main`, and `memos`. The file `memos` is now the current file (because it has just been `opened`). To see it:

```
display ↵
```

1 Type `display` and press **ENTER**
The DISPLAY AREA clears the Microdrive directory from the screen and displays the first record of the `memos` file. Note that the logical name is now `m`.

The first record of the file `memos` is the current record. In the same way, the file `memos` is the current file. It became the current file, in place of `guests`, when it was opened. The command `display` always shows the current file (unless instructed otherwise).

You have done nothing to the file `guests` so it is still open.

Adding notes to the `memos` file

Miranda adds two new notes to the `memos` file using the command `insert`.

```
insert ↵ find suitable Hotel for staying guests F5 make a present list
F5  ESC
```

1 Type the following two notes, remembering to press F5 after each one. If you have forgotten how to use `insert`, refer to the

last chapter, or try to do it using ARCHIVE's prompts and the `help` key, F1.

```
find suitable Hotel for staying guests
make a present list
```

Remember to save the last insertion by pressing F5 before leaving `insert`.

Press F4 (version 2) or ESC (both versions) to leave `insert`.

Deleting records

Miranda deletes the note about Nigel's address book which is now redundant.

```
find "nig" ↵
```

1 Type `find "nig"` and press ENTER
The relevant record appears in the DISPLAY AREA.

```
delete ↵
```

2 Type `delete` and press ENTER
The record vanishes, and the DISPLAY AREA shows the next record in the file.

Be careful when using the command `delete`. It removes the current record from the file currently in use, whether you can see it in the DISPLAY AREA or not. Remember that the display shows the last record `displayed`, which may no longer be the current record, eg. following a `use`. It is not possible to recall a deleted record from temporary memory.

Note: If crucial information is lost, it is possible to find it in backup copies of a file. The use of backups is explained at the end of this chapter.

Opening Files with LOOK

The command `look` also opens data files. It is used in exactly the same way as `open`. The only difference is that files opened with `look` can only be looked at. It is not possible to `insert`, `delete` or `alter` records.

ARCHIVE does not save files opened with `look` onto tape when they are closed, since the copies in memory and on tape must still be identical. The copy in current memory is simply wiped out.

Use `look` to prevent accidental changes to files, and to save time when closing files, because `look` does not waste time saving the files back onto tape.

The memos file now contains the following notes:

```
note$
see Vicar
ask Felicity to be a bridesmaid
ask Jayne to make their dresses
ask Mrs. Glazer to arrange the flowers
don't forget buttonholes
white shoes
buy a dress
has Nigel got a morning suit ?
book Pickwick the photographers
car
phone Sarah - where did they find their caterers ?
how many people can we pack into the Tudor Room at the Dunkery Hotel ?
order a cake
find suitable Hotel for staying guests
make a present list
```

Adding records to the guests file

Miranda is now ready to type some names and addresses. These will be held in the guests file which has been given the logical name main. She wants it to be the current file.

Changing the current file with USE

To change the current file to guests:

use "main" ↵

1 Type use "main" and press **ENTER**
(The file guests was opened automatically with the logical name main.)

Note: use is an exception to the rule that ARCHIVE supplies quotation marks when you need them. use doesn't.

The command use followed by a logical file name makes that file the current file. If you do not know which file is the current file, issue a use command anyway. It does not do any damage to ask ARCHIVE to use a file which is already in use and is the current file.

Current records and files

Most commands in ARCHIVE affect the current record in the current file, if no other file is specified.

ARCHIVE only ever works on one piece of information at a time. Commands which affect files and records affect only one file or record at a time. Commands which affect several records affect each in turn. No matter how many records a file has, or how many files are open, only one record in one file will be the current record.

Imagine using several reference books on a desk at once. Each will have a current page, but only one page is the one you are looking at. The book whose current page is being read is the book currently in use.

ARCHIVE reads records in files in exactly the same way. The current record in each file is the record last moved to in that file. Changes in the current file do not change the current records in the files not in use. The current file is the file last opened, created or used by the commands `open`, `look`, `create`, `use`.

When a file is opened, its first record becomes the current record. If the current file is closed, the open file last in use becomes the current file.

To affect a *record* other than the current record, you must first make it become the current record; so move to that record. It is *not* possible to specify that a command affect a record other than the current record.

To affect a *file* other than the current file, you must make it become the current file. However, it *is* possible to do this as part of the command. To specify that a command affects a file other than the current file use the logical file name. The file affected by the command then becomes the current file.

Example: Using a logical name with a command

The same effect can be achieved in two different ways. The current file both before and after the following examples is `file1`, and the result is that the current record in `file2` is deleted.

Example 1
The file `file1` is the current file.

```
>use "file2"      makes file2 the current file
>delete           deletes the current record
>use "file1"      makes file1 the current file again
```

Example 2
The file `file1` is the current file

`>delete "file2"` deletes the current record in `file2`. The current file is now `file2`
`>use "file1"` makes `file1` the current file again

`insert` ↵ 1 Type `insert` and press **ENTER** to start inserting names into the `guests` file.

The CONTROL AREA now presents you with your available options while the command is `insert`. The **TAB** key (ie. `tabulate`) moves the cursor about the record, from field to field; function keys save or abandon the record when you've finished.

Note that **ENTER** has the same effect as **TAB**. The command `insert` goes on offering blank records for insertion into the file until cancelled by **ESC**.

`John and Susan Heath` ↵
2 Type `John and Susan Heath` and press **ENTER**. The cursor moves to the next line.

`43b Cla` 3 Type `43b Cla`
Miranda pauses, and thinks better of being so formal to her friends. She decides to change 'Susan' to 'Sue'.

Editing a field

SHIFT TAB 1 Hold down **SHIFT** and press **TAB**
The cursor moves back up to the beginning of the `name$` field.

SHIFT ◗ 2 Hold down **SHIFT** and press the ◗ key
The cursor jumps to the beginning of the next word.

SHIFT ◗ 3 Press **SHIFT** and ◗ again to reach the 'S' of 'Susan'.

◗ ◗ 4 Press ◗ twice to position the cursor over the letter 's'.

CTRL ◗ ◗ ◗
5 Hold down the **CTRL** key and press ◗ three times to delete the letters under and to the right of the cursor.

`e` ↵ 6 Type `e` and press **ENTER**
The cursor returns to the beginning of the line with the unfinished address on it.

▼ 7 Press ▼
The cursor jumps to the end of the line.

`rkson Rd` ↵ 8 Finish typing the words `Clarkson Rd` and press **ENTER**.

`London` ↵ `SW16` ↵ ↵ `y` ↵
9 Enter the rest of the address as follows, pressing **ENTER** after each line. Simply press **ENTER** on its own to leave a line blank:

```
address2$       :London
address3$       :SW16
address4$       :
coming$         :y
```

Miranda plans to use the `coming$` field to hold the letter `y` if she knows a guest is coming, `n` when they cannot come, and nothing if she has not yet had a reply, or does not know. This will help her use ARCHIVE to select lists and names for printing.

Note that ARCHIVE does not check the information you enter into a text field. ARCHIVE will not object to two names, or even an address appearing in the `name$` field, It is possible to have ARCHIVE check whatever you type in, but you must program these checks yourself.

The cursor is now opposite `partyof`.

`two` ↵ 10 Type `two` and press **ENTER**
If you have ARCHIVE version 2, ARCHIVE will not let you type letters in a number field. If you have ARCHIVE version 1, the WORK AREA displays the error message `variable not found`.

Entering text in number fields

ARCHIVE does not allow text characters to be entered into a number field. ARCHIVE defines any field which does not end in a dollar sign `$` as a number field.

- In version 2: allows text to be typed, but pressing **ENTER** just moves the cursor back to the beginning of the line. Even pressing **ESC** will not move on to the next field.

  The only way to move on in version 2 is to erase the text, or to type a number – in this case correct the line to read 2 then press **ENTER**.

- Version 1 allows text to be entered, but when **ENTER** is pressed, it leaves `insert` and displays an error message.

  If you got an error message, the error has caused ARCHIVE to abandon `insert`, and the record you were working on has not been saved or inserted into the file.

Recovering from errors in INSERT

Fortunately, (if you have version 1) you do not have to start all over again. You can use the following trick to recover any record you accidentally abandon when using `insert`.

`append` ↵ 1 Type `append` and press **ENTER**
This command takes the information you have just typed into

the fields and inserts (or appends) a new record with these fields, exactly as if you had pressed F5. The `append` command is more fully explained in Chapter 5. The record has now been saved in the file.

`alter` ↵ 2 Type `alter` and press ENTER

To correct the information in the field `partyof`:

↵ (*or* TAB) (*6 times*) 3 Press ENTER or TAB to skip the fields up to `partyof`.

`2` ↵ 4 Type the number `2` into the field `partyof` and press ENTER.

Look at the information you have just entered, and check that it is correct.

If there are mistakes, use SHIFT and TAB to move back up through the fields or ENTER to move down, and correct by overwriting or by using the cursors, erasing and retyping.

In version 1 it is possible to go round and round the record in the same direction using ENTER or SHIFT and TAB. If you press ENTER on the bottom field, the cursor will go to the top field. Try it. The cursor goes back to the beginning of the first field. Now press SHIFT and TAB together. Notice that the cursor moves back round to the last field.

At this point the altered record has not been added to the file. It will not be added unless you tell ARCHIVE to do so.

In version 2 pressing ENTER on the last field saves the record. The only way to go back to the first field is to use SHIFT/TAB together.

Saving the record

The way to save an addition (`insert`) or an `alteration` is essentially the same:

F5 1 When you are ready, press F5, (or, in version 2, press ENTER on the last field).

This saves the information as part of the file in the QL's memory, but does not necessarily save the information on tape yet. This only happens when the temporary memory is full, or the file is closed. Note that F5 can be pressed at any place in any field in the record.

Note: Only press F5 when you have entered all the fields you need in the record. If you press F5 before you mean to, the record you were working on at the time will be inserted into the file incomplete. The only way to change it is to exit from `insert` by pressing ESC and then use the `alter` command. The last record added is the current record, so there is no need to issue a `back` command.

Abandoning the record

ESC

1 If, however, before you save the record, you decide that the whole record was a bad idea, press **ESC** (version 1) or **F4** (version 2). This removes the information completely, as if it had never existed, even though the wrong record is still on the screen.

Adding more guests

Add the next record from Miranda's list:

insert ↵

1 Unless you are still in `insert`, type `insert` and press **ENTER**

Sally Jones and Susan Clark ↵ 3b Academy Mews ↵ Chelsea ↵ London ↵ SW3 ↵ y ↵ 2 ↵ F5

2 Type in the next name and address from Miranda's list, remembering to press **ENTER** after each line, and to save it by pressing **F5**.

3 Insert the remaining records from Miranda's and Joyce's lists (but don't type in their notes). Remember to press **F5** to save each record. Leave fields blank if there is nothing to put in them.

Note: Although Miranda entered 'y' in the `coming$` field for the Heaths and Clarks, as she knew they were definitely coming, leave this field blank for all the other guests, as they have not yet replied. Be careful to type the `party of` numbers in the correct field.

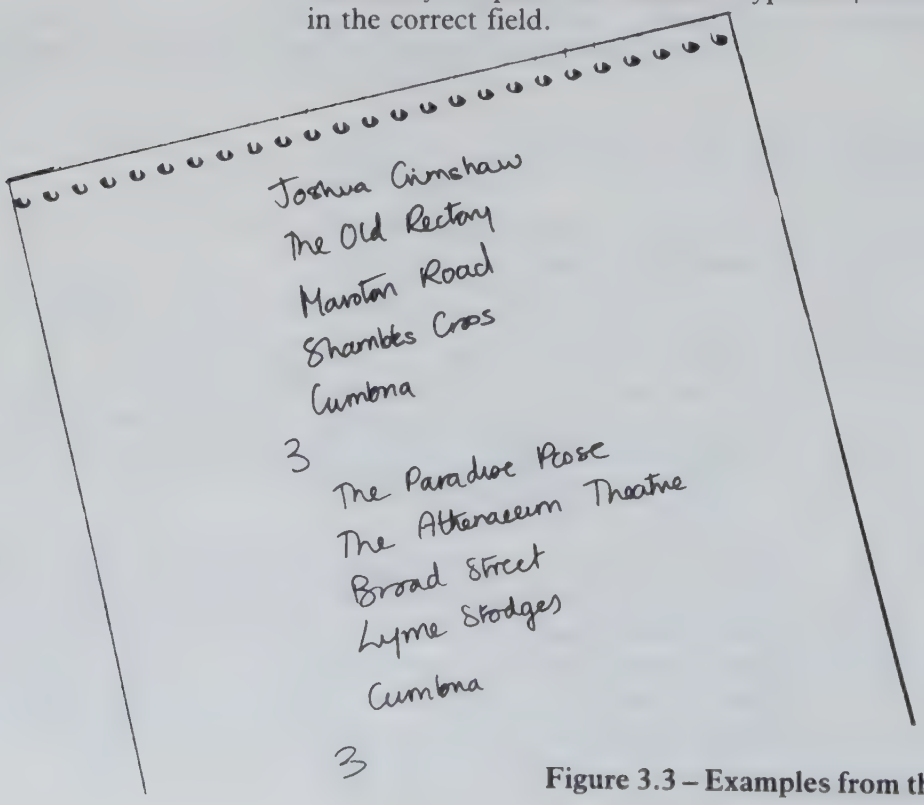
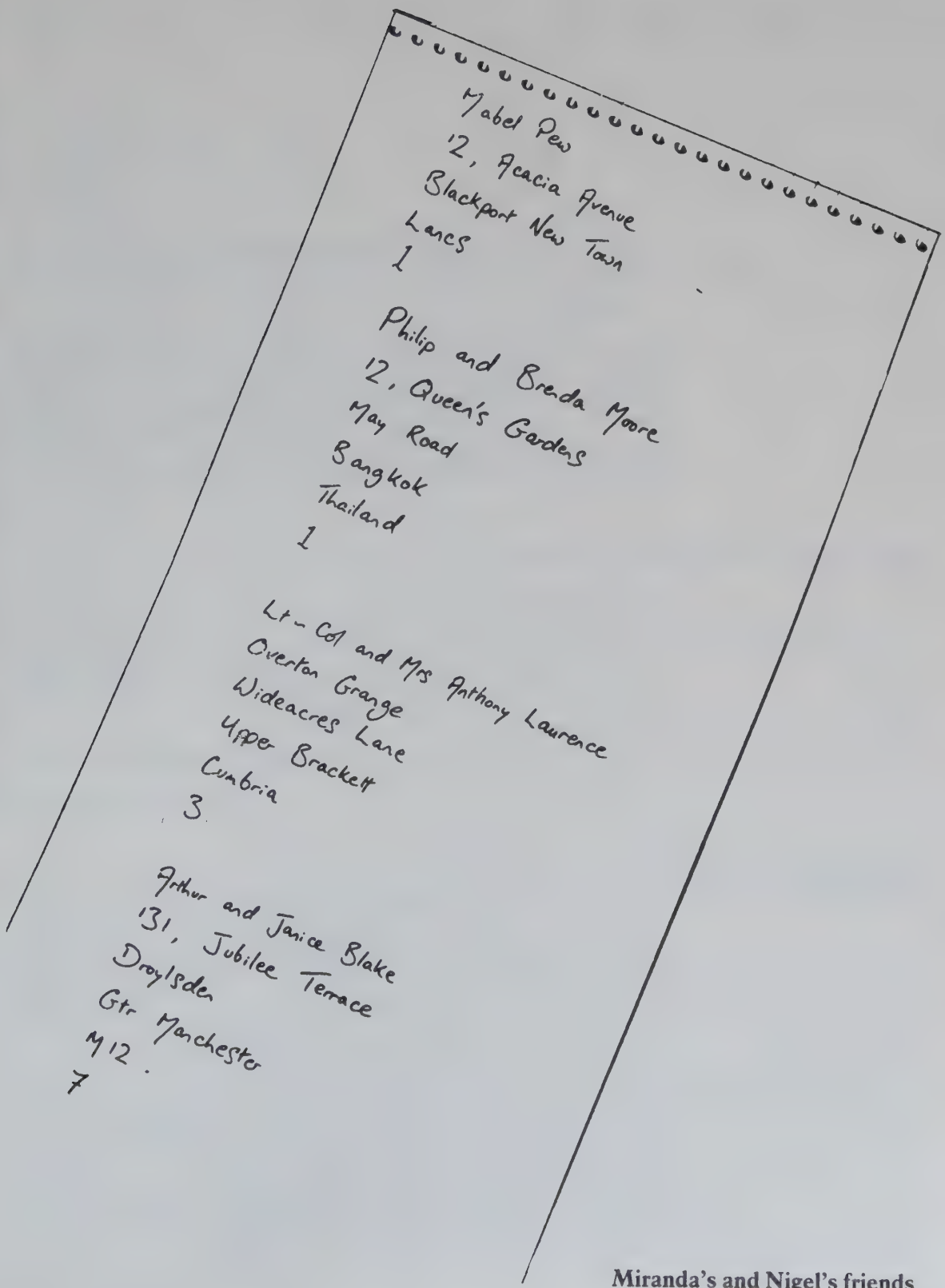
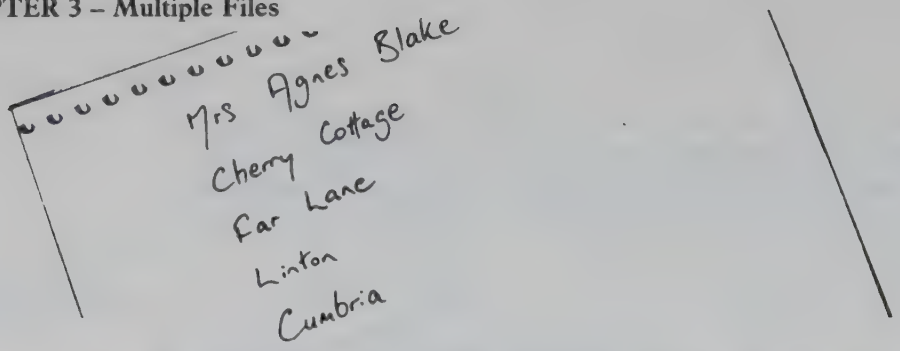


Figure 3.3 – Examples from the Guest List



Miranda's and Nigel's friends



ESC 4 When you have pressed F5 after entering the last record, exit from `insert` by pressing ESC on a blank record (or F4 in version 2).

Note: If you press F5 on a blank record, (or ENTER in the last field in version 2) the blank record is inserted into the file. This is hardly noticeable since nothing on the screen changes, as `insert` presents another blank record in its place. If you find blank records in a file, either fill them in with new information by using `alter`, or remove them from the file completely using the command `delete`.

Displaying the last record added

To view the last record added:

`sprint` ↵ 1 Type `sprint` and press ENTER
The command `sprint` (short for 'screen print') causes the file `displayed` to show the current record. `sprint` has no effect if `display` has not been used first, or if the current file is not the file shown in the `display` layout.

If you think you have the current record displayed, `sprint` will assure you of this.

INSERT and ALTER

Remember that `insert` inserts a new record into the file, and not new information into an existing record. It only adds records, it cannot change them once they have been added. To change an existing record, use `alter` (as explained in the last chapter). `alter` works in exactly the same way as `insert`, except that it only affects one record at a time, the current record.

Note: `alter` and `insert` cannot add or change information in fields which go beyond the screen. In other words, with a 15 line display area, you will be unable to use these commands to affect the fifteenth, sixteenth or higher fields. It can be done by removing the prompts box, designing your own screen(s), or by using other, indirect commands. Avoid the problem by keeping the number of fields fairly small.

The guests file should now contain the following names and addresses:

```
John and Sue Heath              43b Clarkson Road
London          SW16                              y  2
Sally Jones and Susan Clark     3b Academy Mews
Chelsea         London              SW3           y  2
Joshua Grimshaw                 The Old Rectory
Marston Road    Shambles Cross      Cumbria          3
The Paradise Posse              The Athenaeum Theatre
Broad Street    Lyme Stodges        Cumbria          3
Bill Hackney, Drama Critic      The Marston Gazette
22 High Street  Marston on Tees     Cumbria          1
Jon Ducane                      Northern Lights Dance Company
5 Side          Newcastle on Tyne Tyne and Wear      1
Dame Edith Throgmorton-Dunne    The Boundaries
Spiney Wold     Oxon                                 1
Mabel Pew                       12 Acacia Avenue
Blackport New Town Lancs                             1
Philip and Brenda Moore         12 Queens Gardens
May Road        Bangkok             Thailand         2
Lt-Col and Mrs Anthony Laurence Overton Grange
Wideacres Lane  Upper Brackett      Cumbria          3
Arthur and Janice Blake         131 Jubilee Terrace
Droylsden       Gtr Manchester      M12              7
Mrs Agnes Blake                 Cherry Cottage
Far Lane        Linton              Cumbria          1
```

Closing multiple files

Even the most enthralling work eventually palls. After typing in several names and addresses, Miranda saves the work she has done onto tape by closing the file.

close ↵

1 Type close and press ENTER
The right-hand Microdrive whirrs and the last few new records are saved onto tape. The copy of the file in temporary memory is wiped out.

The command close issued without a file name closes the current file, in this case, main. If more than one file is open, the last file in use before the current file becomes the current file. The memos file, m, is now the current file.

Note: the DISPLAY AREA still shows the last record of the guests file. Closing a file does not affect information on the screen, and does not switch off the display. Seeing a file displayed on screen does not necessarily indicate that it is in use, or even open. The display is still showing the last values of the field variables of the

guests file. If you were to reopen the guests file now, the display would automatically change to show the first record in the file.

To close the file memos as well:

close ↵

2 Type close and press ENTER
The Microdrive starts up again, recording the latest version of the memos file back onto tape. The copy of file which was used to load memos into memory from the tape when the file was opened is now being replaced by the revised, larger file.

The command close saves the file onto tape if it was opened with the command open. This is because open allows the file to be added to or changed. If a file is opened with look, no changes are allowed, and the copy of the file in memory is simply discarded when closed.

Closing multiple files

The command close closes the current file if no file is specified. To close a file other than the current file, follow the command with a logical file name, eg. close "main".

If more than one file is open at once, those not in use (ie. not the current file) can only be closed by being specifically named.

If you do not remember the names of the files that are open, simply issue the command close until ARCHIVE displays the error message file not open. You then know that all the files are closed.

All open files are closed by the commands quit or new. Both commands, however, clear absolutely everything from memory, and quit leaves ARCHIVE altogether.

It is better to avoid the situation of forgetting which files must be properly closed by opening as many files as possible using look, and closing files as soon as they are no longer in use. This also frees memory space for other tasks, since each open file is allocated a share of the temporary memory.

Frequent closing of opened files, however, can take up so much time (saving the files onto tape) as to make this last suggestion impractical.

MAKING BACKUP COPIES OF FILES

It is vitally important to make backup copies of your files. The tape in Microdrive cartridges is very vulnerable, and many hours of work may be lost through something as simple as a finger print or a speck of dust. Alternatively, files can be accidentally erased by using the wrong name, or by reformatting a cartridge with files on it. Other possible causes of tape failure are listed in the appendix.

The solution to this threat is to make copies of your files as often as possible.

In ARCHIVE, the command `backup` is used to make copies of files. It only affects files on tape, and will not copy data files if they are still open in memory.

Copying to the same tape with BACKUP

```
backup "guests_dbf" as "guests1_dbf" ↵
```

1 Type `backup "guests_dbf" as "guests1_dbf"` and press **ENTER**
Notice that both `guests` and `guests1` have the file name extension `_dbf`.

The data drive begins to whirr. ARCHIVE is making a duplicate copy of the `guests` file on the same cartridge as the original.

```
dir ↵
```

2 Type `dir` and press **ENTER**

↵

3 Press **ENTER** again to tell ARCHIVE that you want a directory of the data cartridge (mdv2). ARCHIVE assumes you mean mdv2 unless you tell it otherwise (eg. `dir "mdv1_"`).

You should now have three files on the tape: `memos_dbf`, `guests_dbf` and `guests1_dbf`.

```
backup "guests_dbf" as "guests_bak" ↵
```

4 Type `backup "guests_dbf" as "guests_bak"` and press **ENTER**

```
dir ↵
```

5 Ask for another directory listing.
Notice that the file `guests_bak` has appeared in the list. There are now three copies of the file `guests` on the cartridge in Microdrive 2, `guests_dbf`, `guests1_dbf` and `guests_bak`.

Note: be careful when asking ARCHIVE to make a copy file with the same name as a file that already exists. ARCHIVE version 1 could wipe out without warning any existing file with the same name as the copy file is given. As a protection against this, all versions of ARCHIVE will refuse to copy a file to a file with the same name as itself on the same tape. eg. `backup "memos_dbf" as "memos_dbf"` does not work. This is taken further in version 2

when an error message will warn you if you attempt to overwrite a file with a file of the same name. One way of getting round this is to `kill` the old file first (see Chapter 5).

Making a backup copy on the same tape helps protect you from accidental erasure, or from only part of the tape becoming damaged. For better security, make backup copies on a different cartridge.

Formatting a cartridge

You can only use a cartridge for storing files and data after it has been formatted. Formatting merely involves marking out the tape into sectors so that the QL knows where to store and look for information.

- Beware – formatting a cartridge erases everything that exists on that tape. It is safer to do a directory of a tape before formatting.

If you do not have a spare formatted cartridge, use the command `format` to prepare one. This works in a very similar way to the SUPERBASIC command described in Chapter 1.

As a precaution, remove both the ARCHIVE programs and your data cartridge first, leaving your new cartridge in drive 1.

1 Place the cartridge to be formatted in drive 1

```
format "mdv1_arcdatabu1" ↵
```

2 Type `format "mdv1_arcdatabu1"` and press **ENTER**
This formats the tape in drive 1, and gives it the label (or volume name) of `arcdatabu1`, which stands for 'Archive data backup number 1'. You can of course give it any name you want, as long as it does not exceed 10 characters (excluding the drive name), and contains no spaces. Avoid the underline character, '_', as it has special meanings when using the QL in a network.

Once a cartridge is formatted, you do not have to format it again unless you want to clear files from old tapes that you want to reuse for another purpose, or to rescue 'corrupted' tapes. These cause tape operations to fail with error messages like 'bad line' or 'bad medium'. Reformatting the tape can restore it to use by re-laying the sectors to avoid the bad patch.

- Remember – reformatting overwrites everything on the tape.

Don't forget to label the cartridge clearly, using one of the self-adhesive labels supplied in the pack.

Copying to another drive

The (newly) formatted backup tape should be in drive 1.

1 Place the data cartridge back in its usual drive, drive 2.

```
backup "guests_dbf" as "mdv1_guests_dbf"
```

2 Type `backup "guests_dbf" as "mdv1_guests_dbf"` and press ENTER This copies the file on drive 2 onto drive 1, keeping the same name. You should make the backup tape an exact copy of your master data tape, so that if anything happens to the original tape, all the information on it can be restored from the backup copy.

```
dir mdv1_
```

3 Type `dir` and press ENTER, and then type `mdv1_` and press ENTER to check that all the files have been copied.

Note the '_' at the end of the drive name.

Always check you have the right tapes in each drive before you make backup copies. If you leave your copy of the ARCHIVE program tape in drive 1 and haven't removed the write-protect tab, files will be added to it until the tape is full, which happens quite quickly. If you have removed the tab, the files will not be copied, and the commands will fail, sometimes with unpredictable results.

```
backup "memos_dbf" as "mdv1_memos_dbf"
```

4 Make a backup copy of the `memos` file in the same way. Remember to give the full file name and extension, as `backup` makes no assumptions about the file types it is given to copy.

Rotating backups

You should have at least two back up tapes, labelled 1 and 2, for maximum security. It is possible that if a copying operation goes badly wrong that you will lose data on both the original and the tape you are copying to.

Make your first backup on backup tape 1, and the next time use backup tape 2. Then use 1 again, and then 2, and so on. This way, if the worst happens, you will still have a copy which is no older than the last backup you made.

Keep a list and write down the date of each backup and the number of the tape used. The last number on the list is the tape NOT to use when you next make backups of your files,

If you want to use ARCHIVE's HELP facilities, don't forget to put back the ARCHIVE program cartridge in Microdrive 1 before continuing with your work.

Dating your cartridge

The system of rotating backups is effective but depends on knowing which is the oldest backup tape.

It is easy to lose the scraps of paper containing backup dates and tape numbers. If you do, you will have difficulty knowing which backup cartridge to use.

The solution is to date the cartridge itself.

Use the `save` command to create an empty program file named as a date label. There should not be any procedures in memory when you do this. (Programs, procedures and the command `save` are fully introduced in the next Chapter.)

`new` ↵

1 Type `new` and press **ENTER** to clear all procedures from memory.

`save` ↵ `apr6_lab` ↵

2 Type `save "apr6_lab"` and press **ENTER**
The `lab` extension identifies this as the tape label.

F5 `mdv1_` ↵

3 Recall the previous command line using F5, and add `mdv1_` to the beginning of the line. This puts the same date on the backup tape.

Next time you need to make a backup, if the backup cartridge has a `_lab` file with the same date as your working file date, you are probably using the wrong backup cartridge.

Always date both tapes after making backup copies, and delete the old `_lab` files using `kill`.

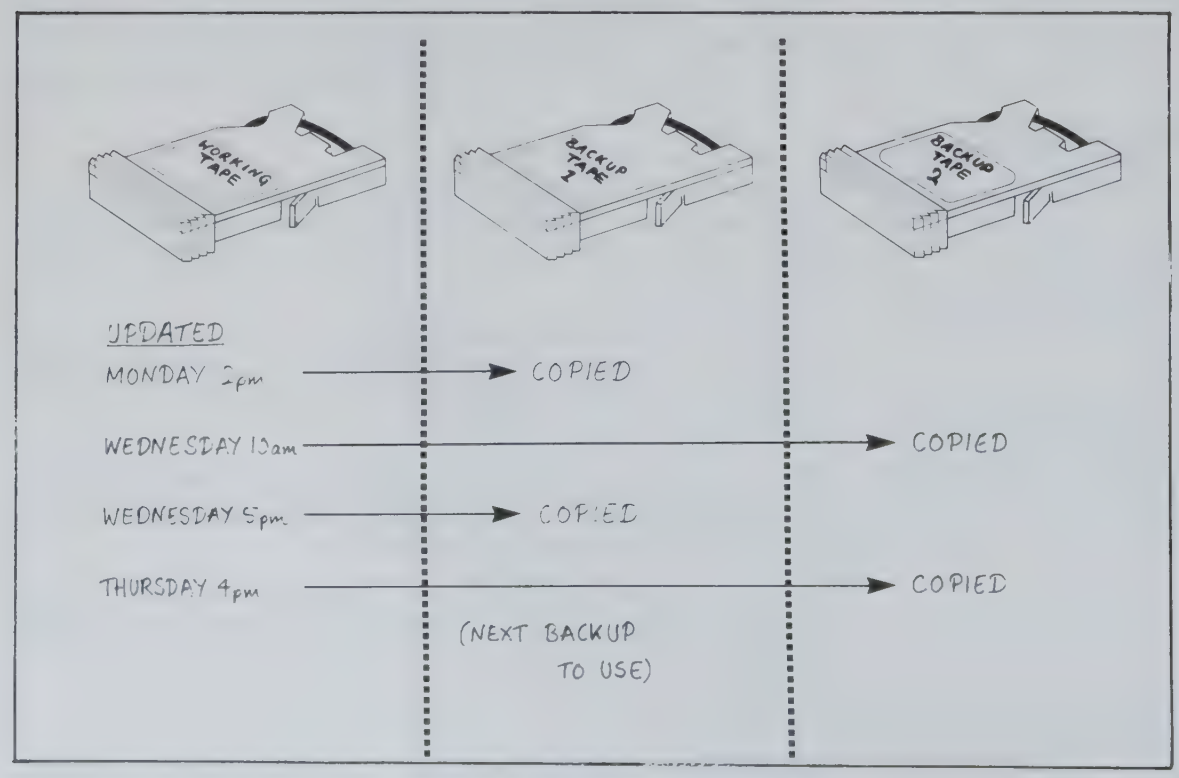


Figure 3.4 – Rotating Backups

SUMMARY

Miranda has created a second file, `guests`, with more than one field in it. She opens her first file, `memos`, using the command `open`. She has to give `memos` a logical file name, as she already has one file open. She inserts some records and deletes a redundant record from the `memos` file using the command `delete`. This command deletes the current record in the current file, unless a logical file name is specified. She then makes the `guests` file the current file, instead of `memos`, by using the command `use` with a logical file name. The command `use` makes a file that is open in temporary memory the current file.

Miranda enters and amends information in her two files using the commands `insert` and `alter`. She then closes her files using the `close` command, and makes `backup` copies both onto the same tape, and onto a new backup tape, which she `formats` first.

Helpful hints

- Write down what you plan to do before you start using ARCHIVE. This will clarify your thinking and be useful as a reference.
- Use `dir` to check that files don't already exist on tape before using `create` and `backup`. Otherwise, ARCHIVE replaces them without warning (version 1).
- Make `backup` copies of your files on separate tapes as often as you can.
- Save your work regularly by closing the data file you are adding to or altering.
- Rotate your backup copies so as never to lose more than a little work, even if a backup copy is destroyed.
- Use the up arrow before inserting a character at the start of a field in `alter`. This prevents ARCHIVE from clearing the line.
- Don't press F5 in `insert` on a blank record. It doesn't show at the time, but the empty record is saved.
- `save` empty files with the backup date in the file name to identify when the backup was made.

CHAPTER 4

Functions and Procedures

The real power of ARCHIVE comes from creating your own commands. These make ARCHIVE work specifically for the purpose you have in mind, and save repeatedly typing a series of commands to perform the same tasks over and over again.

In this chapter, you will learn to use some simple functions and calculations, and to create your own purpose-built lists of commands, called *procedures*, using the line editor.

The following ARCHIVE commands and functions are introduced:

- The ARCHIVE commands:
  - `edit` – edits procedures
  - `print` – prints information on the screen
  - `load` – reads a program file into temporary memory
  - `save` – saves a program file to long-term storage
  - `merge` – combines more than one program file
  - `run` – carries out the instructions in a program
  - `rem` – allows a non-command (remark) line in a procedure

- The ARCHIVE functions:
  - `count()` – shows the number of records in a file
  - `month()` – displays the name of the month
  - `upper()` – converts lower case letters to upper case

Points to remember

- You use a computer to do your work for you. Make use of keys like F5. Create your own commands using the editor.
- The procedure editor is a line editor like those in EASEL and ABACUS.
- `save` your procedures regularly or you may lose them.

- If stuck anywhere in `edit`, press ESC. The most you can lose is the line you are typing.
- Most procedures should perform a single task so that they can easily be used on their own or in other procedures.
- You cannot selectively save procedures, or merge only parts of a program

---

Example: Using the guest list

Getting started

If you do not have the ARCHIVE program loaded, load it now, as described in Chapter 1. Make sure you have the data cartridge with the files `memos_dbf` and `guests_dbf` in drive 2. Use the `dir` command to see which files you have on the tape if you are not sure.

COUNTING THE FILE

The PRINT command and the COUNT() function

Miranda is keen to know how many more names and addresses she has to type to complete her guest list. The addresses on her complete paper notes total 45. She asks ARCHIVE to tell her how many records there are in `guests`.

```
look "guests" logical "g" ↵
```

1 Open the file `guests` with the command `look`

```
display ↵
```

2 Type `display` and press ENTER to display the file

```
print count()
```

3 Type `print count()`
Note that the word `count` is immediately followed by opening and closing brackets, with no spaces. If nothing appears between the brackets, ARCHIVE shows a count of the number of records in the current file.

↵ 4 Press ENTER
The DISPLAY AREA clears and 12, the number of records in the file, appears in the top left hand corner of the DISPLAY AREA. It does not represent the number of people, since `count()` takes no account of how many names appear in any one name field, or of the figure which appears in the `partyof` field. Since there can only sensibly be one address per record, it does represent the number of addresses, and therefore the number of letters that need to be sent.

Displaying the results of COUNT()

On some television sets, the extreme left of the display may be cut off, making the number 12 in the last example appear to be 2.

To display the number starting five columns in:

```
print"     ";count()↵
```

1 Type `print "     ";count()` and press **ENTER**
ARCHIVE displays the number 12 five spaces in.

The command in this line is `print`. The command `print` displays whatever you ask it to on the screen, not the printer.

Note that putting spaces between the quotes causes ARCHIVE to `print` the spaces literally as spaces on the screen.

To print text, use quotes around the text. If there are no quotes, ARCHIVE expects a field name, number, variable or function.

Notice the semi-colon (`;`) separating the two items in the print statement. If you had any more items to `print`, these would also be separated by (`;`). They all appear on the screen on the same line.

Functions

In programming terms, `count()` is called a *function*. Functions, when used in a command line, always produce a result in the form of a number or text. A function is said to *return a value*. Note that this 'value' does not have to be numeric – it is often a string of characters.

For example, other functions 'return' a month from a number, or convert lower case letters into upper case ones:

`print month(2)` displays `February`, the second month

`print upper("word")` displays `WORD`

Functions are not commands: they cannot stand alone in a command line. They have to be used in conjunction with a command, eg. `print count()`.

Functions can always be recognised by the brackets which form the end of the word. Some functions, like `count()` do not have to have anything inside the brackets.

Information inside a function's brackets is technically known as the function's 'arguments'.

To `count()` a file which is open but not in use, type a logical file name between the brackets, eg.
`print count("main").`
Otherwise ARCHIVE assumes you want a count of the current file.

`print name$` ↵

1 Type `print name$` and press **ENTER**
The name field from the current record appears on the line below the count figure in the DISPLAY AREA. ARCHIVE displays:

`John and Sue Heath`

`print 45-count()` ↵

2 Type `print 45-count()` and press **ENTER**
A figure appears below the name representing the number of addresses Miranda still has to type in (45 is the total number of addresses on her list). ARCHIVE displays `33`.

Note that the symbol for a minus sign is the hyphen.

Displaying the result as a decimal fraction

Miranda wants to find out what proportion of her task has been done.

`print count()/45` ↵

3 Type `print count()/45` and press **ENTER**
The new figure in the list represents the number of records in the file divided by the total addresses on paper.

- Note that the symbol for division is the slash, '/'

ARCHIVE displays `0.266666666667` (*just over a quarter*)

Displaying the result as a percentage

To see this as a percentage:

F5 ↓ `*100` ↵

4 Press F5 to recall the last line and add `*100` to the end of it. The line now reads:
`print count()/45*100`
Press **ENTER**.

- Note that the symbol for multiplication in ARCHIVE is an asterisk, '*', not an 'x'.

ARCHIVE displays `26.666666666`

Displaying the cost of postage

`print count()*17` ↵

5 Type `print count()*17` and press **ENTER** to multiply the number of letters sent by the cost of a stamp.

ARCHIVE displays `204`, that is 204 pence.

F5 ↓ ↑ ↑ . ↵

6 Press F5 to recall the last line, and add a decimal point before the 17 to see the result in pounds.

ARCHIVE displays `2.04`.

Arithmetic operators and expressions

The signs for multiplication, division, addition and subtraction (`*`, `/`, `+`, `-`, `^`), are called *operators*. They are used to perform mathematical calculations with numbers, numeric fields or functions which return numbers. They 'operate' on the values on either side of them. For example:

```
print    1+2
print    partyof/2
print    count()*.17
```

Spaces are not necessary on either side of an operator, but may be used for clarity, eg. `print 1+2` is the same as `print 1 + 2`.

A collection of values and operators is called an expression. The following are examples of expressions:

```
1+2
count()/45*100
```

The term expression can refer to just a single value.

`count()` is a numeric expression, as it returns a numeric value.

The symbol ^ (SHIFT + 6) stands for exponention, which means 'raising to the power of', eg.

```
print 2^2    displays 4 (2 squared)
print 2^3    displays 8 (2 cubed)
print 2^4    displays 16 (2^4)
```

LOOKING AT THE GUESTS

Restoring the display – the SCREEN command

Miranda wants to look through her guest list.

To restore the display:

`screen` ↵ 1 Type `screen` and press **ENTER**
The list of figures on the screen is replaced by the standard file display, showing the current record.

The command `screen` calls up the display layout which ARCHIVE still holds in temporary memory.

A standard display is held in memory if any of the commands `display`, `alter`, or `insert` have been used.

If there is no screen layout in memory, the command `screen` has no effect.

It is possible to create your own screen layouts – this is discussed fully in Chapter 10.

Getting the standard display

ARCHIVE puts an image on the *screen* of the standard record display and keeps a copy of this screen image in *memory* whenever the command `display` has been used. The command `display` creates a simple field-by-field layout ranged down the left of the screen with the logical file name at the top.

It does this for whichever file happens to be the current file when the command `display` is used. This screen image remains in memory until replaced by issuing the `display` command with another file, or until the memory is cleared by the command `new`.

The display can be temporarily cleared from the screen by the commands `cls` and `dir`, or by using the `print` command after the > prompt. This does not remove the copy of the screen image from memory and it can be restored to the screen by using the command `screen`.

The command `screen` calls up the screen image already in memory. It has no effect if there is no screen image in memory. The `screen` command acts faster than `display`.

The standard display layout

The standard display layout displays the current values of the field's variables it represents, whether the file is still the current file or not.

This leads to some anomalies and potential confusion, such as:

- Displaying fields from other files which have the same field names
- Showing a record on the screen even after the file has been closed or every single record deleted
- Showing a partly inserted record which was abandoned (*this can be cured by using* `sprint`)

The current record is displayed automatically whenever it changes if commands are used directly after the > prompt.

The commands `insert` and `alter` have the same effect as the command `display`, creating the standard screen image in memory (and on the screen if one does not already exist). However, `insert` does not show the last record in the file but instead presents a blank set of fields ready for a new entry.

It is possible to create your own screen layouts, which can be saved on tape and loaded as you wish, instead of using the standard layout provided by `display`. These remain in memory in exactly the same way until replaced by another or by the standard display, or until cleared by `new`. They are cleared temporarily from the screen by `cls`, `dir` and direct `print` commands, and restored by `screen`. They are more fully discussed in Chapter 10.

Browsing through the file

Miranda makes sure that she's looking at the first record she typed in.

`first` ↵ 1 Type `first` and press **ENTER**

Miranda browses through the file looking at the addresses she has **ENTER**ed using `next`, and occasionally `back`, and **F5** to repeat commands. She quickly tires of typing in these words over and over again when she changes from one command to another.

Trying to type faster, she makes mistakes. ARCHIVE dutifully reports that it does not recognise the commands `bext` and `nack`. She types them in again. And occasionally, again.

CREATING COMMANDS (PROCEDURES)

Creating a procedure to move forwards through the file

She decides to speed things up by creating her own commands, or procedures, to do the same thing. They will be only one letter long – `b` for `back`, and `n` for `next`.

`edit` ↵ 1 Type `edit` and press **ENTER**
The DISPLAY AREA clears, and the prompt and the word `edit` vanish from the WORK AREA. Instead, the word `proc` has appeared on the input line, with the cursor waiting after it.

From now until the > prompt reappears, the `edit` command is in control of everything that ARCHIVE does. The > prompt only reappears when you are finished with `edit`. To leave `edit`, press **ESC** until the > prompt reappears.

Note: Unlike elsewhere in ARCHIVE, pressing the **ESC** key in `edit` does not always cause what you are doing to be abandoned. In `edit`, the **ESC** key is used to cancel commands and to leave `edit` itself without losing anything. The only time **ESC** abandons information is when typing a line in the WORK AREA. This will become apparent as you learn to use `edit`.

The CONTROL AREA reads:

```
CREATE a new procedure. Enter the procedure name
```

ARCHIVE calls a command that you create yourself a *procedure*. You are being asked to tell ARCHIVE the name you have chosen for the procedure.

Note: If you make a typing mistake while typing in a line, correct it using the usual editing keys. The cursor keys, backspace and others work in `edit` in exactly the same way as they do when you are typing in a field or a command.

`n` ↵ 2 Type `n` and press **ENTER**
The WORK AREA clears, and two lines appear in the DISPLAY AREA:

```
n       proc n
          endproc
```

The words `proc` and `endproc` are supplied by ARCHIVE, and cannot be changed. The procedure will not work without them. Notice the way ARCHIVE indents the lines. You will discover that this helps you to keep track of where you are going in more complex routines.

The `n` in the column on the left is the procedure name. The text in the main part of the DISPLAY AREA represents the contents of the procedure itself.

The CONTROL AREA is prompting you to insert a line. The cursor is waiting in the WORK AREA.

`next` ↵ 3 Type `next` and press **ENTER**
The input line again vanishes from the WORK AREA, and reappears in the DISPLAY AREA between the `proc` and `endproc` lines:

```
proc n
  next
  endproc
```

The line `next` has been inserted into the procedure.

ESC 4 Press **ESC** to stop inserting lines in `edit`.
Note that the prompts box changes as you stop the insert mode.

ESC 5 Press **ESC** to leave `edit`. The DISPLAY AREA clears, and the > prompt reappears. You are no longer under the control of the `edit` command, and ARCHIVE is waiting for another command.

`screen` ↵ 6 If the screen is blank, type `screen` and press **ENTER**. The record you were last looking at reappears. The command `edit` does not affect anything ARCHIVE is doing when you use it. It simply allows you to edit and create procedures, (your own purpose-built commands).

Procedures

A procedure is one or more ARCHIVE commands bundled together and given a name. This name can be used exactly like one of ARCHIVE's commands to perform an action or series of actions. Any command which can be typed in on the input line will work in exactly the same way inside a procedure.

Procedures are used in exactly the same way as commands. Type in the procedure name after the > prompt and press ENTER. ARCHIVE then performs the commands it finds in the procedure in sequence, exactly as if you had typed them one after the other.

When you press ENTER after typing in a word on the input line, ARCHIVE checks its own commands to see if it matches one of them. If it doesn't, ARCHIVE tries to match it with the name of a procedure you have created, if there are any in memory. If there are, and one matches, ARCHIVE executes any instructions it finds between the lines `proc` and `endproc`. Otherwise the error message `command not recognised` appears.

- A procedure name may contain up to 13 characters, must start with a letter, and must contain no spaces. It must not be the same as one of ARCHIVE's key words.
- A procedure may contain up to 255 lines of commands
- You can alter procedures even after you have created them

Procedures can be stored on tape for later use. When you create them, they do not become a permanent part of ARCHIVE's commands. They must be saved onto tape whenever created or changed, or they will be lost every time the QL's memory is cleared. They must be 'loaded' every time you want to use them.

Using your own procedures

n ↵ 1 Type n and press ENTER
The display changes and shows the next record, exactly as if you had typed in `next`.

n ↵ 2 Try it again. You can now move through the file as quickly as by using F5, but can use other commands in between, and are far less likely to mistype the command.

A second procedure to move backwards through the file

To create the abbreviated version of `back`, use the command `edit` to call up the 'procedure editor' again.

The Procedure Editor

The command `edit` provides a *text editor* specifically for writing procedures. A text editor edits text like the QL's word processing program QUILL. Text editors allow text from single words to entire blocks – to be created, changed and deleted. The main difference between `edit` and QUILL is that `edit` is a line editor, which edits only one line at a time.

The command `edit` allows procedures to be created, edited and deleted. It allows lines to be inserted into, changed and moved about in and deleted from procedures. All the usual functions of the editing keys, and function keys F1 and F2 are available.

ARCHIVE's text editor helps write procedures by supplying start and end commands, indenting lines to clarify structure, and checking that the lines are correct (up to a point). Note that quotation marks are not automatically supplied where needed when you are writing procedures.

ARCHIVE sorts the procedures into alphabetical order by name.

It is also possible to write procedures for ARCHIVE using other text editors.

`edit` ↵ 1 Type `edit` and press ENTER

Again, the DISPLAY AREA clears, and the prompt and the word `edit` vanish from the WORK AREA. This time, however, you are not being prompted to create a new procedure.

If there are any procedures already created and in memory, ARCHIVE shows them first. The command `edit` only assumes you are going to create one if none exist.

The procedure `n` is displayed on the screen, and the CONTROL AREA offers a range of options. Those in the large central box refer mainly to editing (ie. changing) existing procedures. The option to create a new procedure is not apparent. To see more of the available options:

F3 2 Press F3, as prompted by the top right hand box. The CONTROL AREA changes, and now shows that four further editing com-

mands can be called by pressing a single key. There is no need to press **ENTER** after pressing one of the four.

n 3 Press n for a new procedure
The screen is now like the first edit screen you see when there are no procedures in memory (except that existing procedures are listed in the DISPLAY AREA). The CONTROL AREA is prompting you to enter a name, and proc is waiting in the WORK AREA.

b ↵ 4 Type b and press **ENTER**
The CONTROL AREA changes to prompt you to insert a line. The WORK AREA clears, and the DISPLAY AREA now shows:

```
b        proc b
n          endproc
```

The column on the left shows the names of all the procedures currently in memory. One of them, the current procedure, is displayed on the right.

ARCHIVE has created the procedure b, and inserted it in the list of procedures on the left of the screen. The list is in alphabetical order. The letter b appears highlighted because it is the current procedure.

The contents of the current procedure are displayed on the right. The line you have just typed, proc b, is highlighted. It is the current line.

The *current line* and *current procedure* are similar concepts to the current record and the current file. They are the ones you have last created or moved to, and are the ones which will be affected by any actions you take.

black ↵ 5 Type black and press **ENTER**
The input line again vanishes from the WORK AREA, and is inserted into the procedure:

```
b             proc b
n               black
                endproc
```

Note that the word black is highlighted. It is now the current line. ARCHIVE is waiting for you to insert more lines. To correct the current line, you must tell ARCHIVE that you do not wish to insert any more lines.

ESC 6 Press **ESC**
The box in the CONTROL AREA changes back to the main edit prompts.

You can also leave the insert mode by pressing **ENTER**, or the up or down arrow keys, when the WORK AREA is blank.

F5 7 Press F5
The word `black` reappears in the WORK AREA. You are now using the *line editor*.

◆ CTRL ◆ 8 Change `black` to `back`. Edit the line using the normal editing keys.

↵ 9 Press ENTER when ready
The WORK AREA clears and the corrected line appears in the procedure in the DISPLAY AREA. You have finished creating the procedure `b`.

The procedure now looks like this:

```
proc b
  back
  endproc
```

Note: If ARCHIVE does not respond to function keys in `edit`, you have probably forgotten to leave the insert mode. Press ESC.

Moving to another procedure

To see the contents of the procedure `n`:

TAB 1 Press the TAB key
The DISPLAY AREA changes to show the first procedure you wrote. The letter 'n' is highlighted in the left margin, and the first line of the procedure is the current line.

Notice that although you wrote `n` first, ARCHIVE has put the procedures into alphabetical order.

Pressing TAB goes forward one procedure, pressing SHIFT and TAB together goes back one procedure. The phrase `INV+TAB` (in the CONTROL AREA) is a mistake, and should read: `SHIFT+TAB`.

To leave `edit`:

ESC 2 Press ESC

Try out the new commands `n` and `b`.

Cutting Lines

If you accidentally insert an extra line when creating a procedure, it can be deleted by leaving the insert mode and pressing F3 followed by `c`, which stands for *cut*. Unlike F3 followed by `n`, this does not take effect immediately. You are prompted to confirm that you wish to cut the current (highlighted) line by pressing ENTER. If you do not wish to cut the current line, press ESC.

To leave `edit`:

ESC 2 Press ESC

Try out the new commands `n` and `b`.

Inserting lines

As long as the prompt `INSERT a line` is visible in the CONTROL AREA, you can go on typing in lines in the WORK AREA and inserting them in the procedure by pressing ENTER. The new lines appear immediately below the current line. The current line always appears highlighted or white. Each line in turn becomes the current line as it is inserted.

In ARCHIVE's terms you are in *insert* mode. This means that all you can do is insert lines. It is similar to the command `insert` in as much as you cannot change what you have inserted without first moving out of the insert mode. While in the insert mode, you can change a line until you press ENTER.

ARCHIVE automatically puts you in insert mode when you create a new procedure; obviously, there can be no existing lines to edit. At other times press F4 to start inserting lines. This will only work when you can see the main edit prompt in the control box, including the F4-insert mode.

To stop inserting lines, you must leave the insert mode. You cannot edit a previous line, or use any of the commands available with F3, while in insert mode. To leave insert mode, press ESC, (or press ENTER or either of the up and down cursor keys while on a blank line). Any line that you are working on which has not been ENTERed will be abandoned and lost when you press ESC.

Always press ENTER to insert a line. To re-start the insert mode after pressing ESC, press F4.

Creating more procedures

Use the `edit` command to create simple procedures using the commands `display`, `alter`, `insert` and `edit`, giving them the first letter of each command as names, in the same way as `next` and `back`.

`edit` ↵ 1 Type `edit` and press ENTER

F3 2 Press F3

`n` 3 Press `n` to start a new procedure

`d` ↵ 4 Type in the new procedure name

`display` ↵ 5 Type in the command for the procedure

↵ 6 After inserting a line, remember to press **ENTER** to leave the insert mode. Pressing **F3** or **F5** while in the insert mode has no effect.

F3 `na` ↵ `alter` ↵ ↵ F3 `ni` ↵ `insert` ↵ ↵ F3 `ne` ↵ `edit` ↵ ↵
The procedures should look like this:

```
proc a
  alter
  endproc
proc b
  back
  endproc
proc d
  display
  endproc
proc e
  edit
  endproc
proc i
  insert
  endproc
proc n
  next
  endproc
```

ESC 7 Press **ESC** to leave `edit`.

Saving copies of procedures – the SAVE command

It is important to remember that while data files are saved automatically when you close them, procedures must actually be *saved*: ARCHIVE doesn't prompt you to do this, so its easy to lose procedures accidentally.

Procedures can be lost by using `new` or `quit`, switching off, or loading other procedures from tape.

Before going any further, Miranda saves a copy of her procedures on to tape in a file called `easy`. So far, they have only been created in the temporary memory. If the power fails or the memory is cleared for any reason, they will be lost.

`save` ↵ `easy` ↵ 1 Type `save` and press **ENTER** and then `easy` and press **ENTER**. The right-hand microdrive whirrs, and all the procedures are saved onto tape as a file with the name `easy_prg`. This is a *program file*, as opposed to a data file. Unlike closing and saving data files, which clears the data, saving procedures does not 'close' them or clear them from the temporary memory (RAM).

In ARCHIVE's terminology, there is a distinction between a procedure and a program. A program is a collection of procedures brought together to perform specific tasks.

It is a good idea to keep programs and procedures as short as possible. This uses less memory, and makes editing programs more manageable.

`dir` ↵ ↵ 2 Have a look at the directory of drive 2 – `easy_prg` will appear in the list if it has been saved correctly. Your tape should now contain the following files: `memos_dbf`, `guests_dbf`, `guests1_dbf`, `guests_bak` and `easy_prg` and a date label file such as `dec9_lab`.

A STARTING PROCEDURE TO OPEN FILES

Miranda creates a procedure which will open the files she writes each time she comes into ARCHIVE.

She starts a fresh program to contain the necessary procedures, because they will be specifically connected with the wedding files, and will be stored under a different name to distinguish them from the `easy` procedures which could be used with any file.

She plans to write procedures to `open`, `close` and `backup` files for her, as the commands used are long, riddled with quotation marks and easy to mistype.

Open `guests` file and `memos` file

She gives some of the commands apparently meaningless but brief names, such as `og` to stand for `open "guests"`.

These will not be easy to remember out of the context of the wedding files, but will be easy to use.

`new` ↵ 1 Type `new` and press **ENTER**
If you opened your data files with `look`, these will be quickly cleared from memory. If you opened them using `open`, they will be saved, with any changes you have made, back onto tape. Any display on the screen will be cleared, as will all the procedures currently in memory.

Always make sure you save your procedures, if they are new or have been changed, before issuing the `new` command.

`edit` ↵ 2 Type `edit` and press **ENTER**
As no procedures exist in temporary memory, ARCHIVE is immediately ready for you to type in a new procedure name.

`og` ↵ 3 Type `og` and press **ENTER**
ARCHIVE automatically switches on the insert mode.

print "Opening Guests File"↵

4 Type `print "Opening Guests File"` and press **ENTER**

open "guests"logical "g"↵

5 Type `open "guests"logical "g"` and press **ENTER**

ESC 6 Press **ESC** to leave the insert mode

The procedure `og` should look like this:

```
proc og
  print "Opening Guests File"
  open "guests"logical "g"
  endproc
```

F3 7 Press **F3**

n 8 Press `n` to start another procedure.

om↵print "Opening Memos File"↵open "memos"logical "m"↵

9 Call the procedure `om` and type in two lines:

```
print "Opening Memos File"
open "memos"logical "m"
```

Note: Use `print` commands to let you know on the screen what is happening. If one in a series of procedures fails, it will be easy to see what has and hasn't been done.

ESC 10 Leave insert mode by pressing **ESC**

The procedure `om` should look like this:

```
proc om
  print "Opening Memos File"
  open "memos"logical "m"
  endproc
```

A procedure to open both files

F3 n 1 Start a new procedure by pressing **F3** and `n`.

open↵ 2 Type `open` and press **ENTER**
Nothing happens. ARCHIVE refuses to accept one of its own commands as a procedure name.

both↵ 3 Add the letters `both` to the name, making it `openboth`, and press **ENTER**.

om↵ 4 Type the first line as `om` and press **ENTER**
Remember that you are still in the insert mode. You can go on inserting lines by typing them in and pressing **ENTER** until you leave the insert mode. Each line will appear in the WORK AREA as you type it.

og↵ 5 Type `og` and press **ENTER**

Your procedure should look like this:

```
proc openboth
  om
  og
  endproc
```

ESC 6 Press ESC to leave the insert mode. The line above the last line is highlighted. It is now the current line.

↑ 7 Press the up arrow. The line above becomes the current line.

↑↓↑ 8 Press the up and down arrows to see how they work. Notice that you cannot make `endproc` the current line. It is not possible to change or delete `endproc`. It *is* possible to edit the top line, but not the word `proc` itself.

Note: You can change the procedure name by editing the top line.

Notice that any one of the procedures you have just created can be used on its own. Alternatively, the two together can be called by typing in `openboth` at the command line.

Procedures which contain the names of other procedures as commands are said to *call* another procedure into use. Its commands are then executed before returning control to the calling procedure.

Procedures to close each file

Miranda creates some more sets of procedures in the same way as before.

F3 n cm ↵ 1 Create a procedure called `cm` (to close the `memos` file).

```
print "Closing Memos File" ↵
```

2 First, tell ARCHIVE to print a message saying which file is being closed. Type `print "Closing Memos File"` and press ENTER.

```
close "m" ↵
```

3 Now tell ARCHIVE to close the `memos` file, by typing `close "m"` and pressing ENTER. `m` is used here because `memos` is opened with the logical name `m`.

↵ 4 Press ENTER to leave insert mode.

The procedure should look like this:

```
proc cm
  print "Closing Memos File"
  close "m"
  endproc
```

F3 n cg 5 Now press F3 and n to create a similar procedure to close the `guests` file, called `cg`.

print "Closing Guests File" ↵

6 Type `print "Closing Guests File"` and press **ENTER**.

close "g" ↵

7 Type `close "g"` and press **ENTER**.

↵

8 Press **ENTER** to leave insert mode.

The procedure `cg` should look like this:

```
proc cg
  print "Closing Guests File"
  close "g"
  endproc
```

A procedure to close both files

F3 n closeboth ↵

1 Now create a procedure called `closeboth` to call (ie. set into operation) the two procedures you have just created, `cm` and `cg`.

cm ↵

2 Type `cm` and press **ENTER**
This calls the procedure `cm` which closes the `memos` file.

cg ↵

3 Type `cg` and press **ENTER**
When ARCHIVE has finished carrying out the commands in `cm`, it returns control to `closeboth`. This then calls the procedure `cg`.

↵

4 Press **ENTER** to leave insert mode

The procedure `closeboth` should look like this:

```
proc closeboth
  cm
  cg
  endproc
```

A procedure to backup both files

Now create three more procedures to backup the `memos` file and the `guests` file.

F3 n

1 Press **F3** and `n` to create a new procedure.

bm ↵

2 Call the procedure `bm` (back up the `memos` file).

print "Backing up Memos File" ↵

3 Print a message on the screen to say which file ARCHIVE is copying.

backup "memos_dbf" as "mdv1_memos_dbf" ↵

4 Tell ARCHIVE to copy the `memos` file to drive 1.

↵

5 Leave insert mode.

The procedure bm should look like this:

```
proc bm
  print "Backing up Memos File"
  backup "memos_dbf"as "mdv1_memos_dbf"
  endproc
```

F3 n bg ↵ print "Backing up Guests File" ↵ backup "guests_dbf" as "mdv1_guests_dbf" ↵

6 Create a second procedure like bm, but make it copy the guests file to drive 1. Don't forget to print a message on the screen saying which file is being copied.

↵ 7 Leave insert mode by pressing ENTER.

The procedure bg should look like this:

```
proc bg
  print "Backing up Guests File"
  backup "guests_dbf"as "mdv1_guests_dbf"
  endproc
```

F3 n backupboth ↵ bm ↵ bg ↵ ESC

8 Finally, create a procedure to call the two procedures bm and bg, called backupboth.

The procedure backupboth should look like this:

```
proc backupboth
  bm
  bg
  endproc
```

These procedures are limited to use with the wedding data files, and will be superceded as you develop your programs. However, they will be useful in removing some of the tedium and error connected with operations involving the microdrives.

A backup procedure

To make backup copies in the middle of a long session, you will need to close the data files, back them up to drive 1, and then re-open them before carrying on.

F3 n makecopies ↵ closeboth ↵ backupboth ↵ openboth ↵ ESC

1 Create a procedure to do this for you. It should look like this:

```
proc makecopies
  closeboth
  backupboth
  openboth
  endproc
```

Remember that a bad tape can cause tape commands to fail, and that you cannot close files which are not open. If any one command fails, all those following it will not be executed.

Note: The command `new` closes all data files and clears the memory, but cannot be used inside a procedure as it clears the rest of its own procedure from memory as well.

PUTTING YOUR PROCEDURES TOGETHER

The START procedure and the RUN command

F3 `n start` ↵ `REM WEDFILES_PRG 5 April 1984` ↵ `REM procedures to open, close and copy data files` ↵ `openboth` ↵ ESC

1 Finally, create a procedure called `start` which looks like this:

```
proc start
  REM   WEDFILES_PRG  5 April 1984
  REM   procedures to open, close and copy data files
  openboth
  endproc
```

Procedures called `start` have a special significance in ARCHIVE. The command `run` followed by a program name will load that program's procedures and look for the one called `start`, which it will execute automatically.

Notice the use of `rem` to identify the program in use, and the date it was last changed. This is an abbreviation for 'remark' and when ARCHIVE comes across it in a procedure, it ignores anything after it on the same line.

ESC 2 Leave `edit` by pressing **ESC**.

Note: You cannot use the command `run` with a program or set of procedures which does not contain a procedure called `start`. The procedures will be loaded, but the error message `command not recognized` appears.

Bringing procedures into current memory – the MERGE command

`merge` ↵ `easy` ↵ 1 Bring the `easy` procedures into temporary memory so that they can be used alongside the procedures you have just created. Do this by using the command `merge`. Type `merge` and press **ENTER**, then type `easy` and press **ENTER**.

The command `merge` tells ARCHIVE to merge the procedures found in the named program file with any already in memory. If

ARCHIVE finds that any procedures have the same name, those from the file being merged replace those already in memory.

Be careful not to lose procedures this way.

This command should not be issued from within a procedure (in version 1), because it can cause errors.

Saving procedures

Save the new procedures onto tape with the name `wedfiles`.

```
save↵wedfiles↵
```

1 Type `save` and press **ENTER**, and then `wedfiles` and press **ENTER**. The procedures are saved onto the tape in microdrive 2 with the name `wedfiles_prg`.

Making a backup copy

Put the back-up cartridge into microdrive 1 to make a copy of the procedures you have just written. Use **F5** to recall the previous line, and add `mdv1_` to the beginning of the file name, so the procedures are saved to drive 1 with the same name as drive 2.

```
F5  CTRL  ↑  ↑  mdv1_
```

1 Add `mdv1_` to the file name and press **ENTER**.

This is the easiest way to make backup copies of program files you are working on. You can of course, use the `backup` command to do this. If you use `backup`, remember to use the `_prg` extension with the filename. The `backup` command will copy a program file even if the procedures are open in memory.

Note: Like the `create`, `close` and `backup` commands in version 1 the `save` command wipes out any file of the same name that it finds on the tape.

For this reason, it is a good idea to save your programs periodically with different names as you develop them. For example, name successive generations `procs1`, `procs2`, `procs3` etc. until you are sure they are correct. Save the final version as `procs`. This way, if you make changes you do not like, you still have the old version.

`wedfiles_prg` now contains the following procedures; use the `tabulate` (or **SHIFT** + **TAB**) key to move up and down the list. Chapter 6 explains how to list your procedures on paper, if you have a printer.

Note: You can't `LLIST` to the screen unless you use `SPOOLON SCREEN`, and anyway, there is no need.

The procedures in `wedfiles_prg`

The procedures now look like this:

```
proc a
  alter
  endproc
proc b
  back
  endproc
proc backupboth
  bm
  bg
  endproc
proc bg
  print "Backing up Guests File"
  backup "guests_dbf"as "mdv1_guests_dbf"
  endproc
proc bm
  print "Backing up Memos File"
  backup "memos_dbf"as "mdv1_memos_dbf"
  endproc
proc cg
  print "Closing Guests File"
  close "g"
  endproc
proc closeboth
  om
  og
  endproc
proc cm
  print "Closing Memos File"
  close "m"
  endproc
proc d
  display
  endproc
proc e
  edit
  endproc
proc i
  insert
  endproc
proc makecopies
  closeboth
  backupboth
  openboth
  endproc
proc n
  next
  endproc
proc og
  print "Opening Guests File"
  open "guests"logical "g"
  endproc
proc om
  print "Opening Memos File"
```

```
    open "memos"logical "m"
    endproc
  proc openboth
    om
    og
    endproc
  proc start
    REM   WEDFILES_PRG  5 April 1984
    REM   procedures to open, close and copy data files
    openboth
    endproc
```

`dir` ↵ ↵ 1 Do a directory to see that `wedfiles` has been saved to drive 2. The tape should now contain the files: `memos_dbf`, `guests_dbf`, `guests1_dbf`, `guests_bak`, `easy_prg` and `wedfiles_prg`, and the date label.

A GENERAL PURPOSE BACKUP

The INPUT command

The importance of backing up programs cannot be overestimated. Backing up is tedious, and the easier it can be made, the more you will actually do it. This section develops a general backup procedure for use at any time, in any situation.

You will rapidly outgrow the simple labour-saving procedures you have just written, especially as you continue to create new data and program files. The following procedure asks you for the name of any file you want to backup, and copies it with the same name onto drive 1. Make sure you have the correct backup microdrive in drive 1.

Add the procedure `bu` to your armoury. It will prompt you to **ENTER** a file name and then supply the necessary command to ARCHIVE itself.

`edit` ↵ 1 Type `edit` and press **ENTER**

F3 `n bu` ↵ 2 Press **F3** and `n` to create a new procedure, and call it `bu`.

```
input "Full file name to copy?  "; fn$ ↵
```

3 Type `input "Full file name to copy?  "; fn$` and press **ENTER**. (Notice the space between the ? mark and the closing quotation marks.)

The `input` command takes the form `input "message";variable`. It waits for you to type information on the screen. It assumes you have finished when the **ENTER** key is pressed. It prints any message you put in quotes after it as a prompt, to remind you what you are supposed to be typing in.

Notice that the prompt here reminds you both to use the full file name, and that the name of the file to be copied is needed, rather than the file it is to be copied to.

The information you type in is then stored in memory under the name following the `input` command, in this case `fn$`. This is called a *variable*, since its contents vary according to what you type in.

```
backup fn$ as "mdv1_"+fn$ ↵ ESC
```

4 Now type the line which copies the file name just ENTERed to drive 1.

Notice that even if you don't type a space between `fn$` and `as`, ARCHIVE automatically inserts one in `edit`. ARCHIVE knows that in a command the dollar sign only appears at the end of variable names, so the next letter must be the start of a new word.

Notice the use of a plus sign `+`, to add the variable name to the letters `mdv1_`. Notice also that whereas `"mdv1_"` is in quotation marks, `fn$` is not.

Using a VARIABLE

The variable in this case is `fn$`, which stands for file name, and ends in a dollar sign '$' because file names are *strings of characters* rather than numbers. It is being used to store the file name you type in so that it can be used by the following `backup` command.

The `backup` command normally requires that file names appear between quotes. ARCHIVE then reads the characters between the quotes to find out which files to use. If the word does not appear between quotes, and ends in a dollar sign, ARCHIVE knows the word is a variable. It looks in memory for a variable with that name, and uses the characters stored there as the file name.

In this case, the two file names needed by the `backup` command must be different, so the second one has a literal string of characters, `mdv1_`, added to the front of the variable characters to make a new, longer file name. It instructs ARCHIVE to copy the file onto the cartridge in drive 1, the left-hand drive.

The procedure should look like this:

```
proc bu
  input "Full file name to copy? "; fn$
  backup fn$ as "mdv1_"+fn$
  endproc
```

ESC 5 Leave `edit`.

Now try running this backup procedure:

`bu` ↵ 6 Type `bu` and press **ENTER**
The message `Full file name to copy ?` appears in the DISPLAY AREA, with the cursor waiting after it.

ARCHIVE is waiting for you to type something in.

Make sure you have a backup tape in drive 1

`easy_prg` 7 Type `easy_prg`. The letters you type appear next to the prompt message in the DISPLAY AREA.

↵ 8 Press **ENTER**
The file is now copied from drive 2 to drive 1.

Variables

Variables are exactly like field variables (see Chapter 2), but are not connected with a record or file. Variable names remain the same, but their contents vary according to what you choose to put in them.

Variables are used to store characters and numbers in temporary memory, but not in a record or file.

Variable names:

- must begin with a letter
- may be up to 13 characters long
- may contain no spaces.

If the name ends in a dollar sign, the contents may be any letter or number. These are called *string variables*, as they contain strings of text.

If the name does not end in a dollar sign, the contents must be a number. The signs for the decimal point and minus (`.` and `-`) are allowed. Trying to store anything else will cause an error. These are called *numeric variables*.

You cannot use an ARCHIVE key word as a variable name, eg. `next$` and `logical$` are unacceptable.

Within these limits, you can of course give a variable any name you like, but it helps if it reminds you of what it is being used for, and if it is short, to save on typing and keep programs compact.

Note: It is surprising how easy it is to forget to remove the ARCHIVE program cartridge before doing a backup. If it has been protected by having its write-protect tab removed, this will waste a lot of time and quite possibly 'hang' the system. If unprotected, the tape will soon fill, and you will again waste time.

Warning: If this procedure does not work for you, it may be because some versions of ARCHIVE will not overwrite a file onto a file of the same name. To deal with this, amend your procedure 'bu' by adding the line '`kill "mdv1_"+fn$`' *before* the line beginning `backup`. The `kill` command removes any file of the same name from Microdrive 1 before the backup is made. See the next chapter for further details.

Adding strings of characters

Note that is is possible to add together strings of characters using the plus sign (+). For example, `print "Mr. "+name$+" Esq."` might display `Mr. John Wooten Esq.` Any character string or variable may be added in this way. The process is called *string concatenation* in programming jargon.

Messages and reminders

The bu procedure as it stands works, but can be vastly improved by being more communicative.

`edit` ↵ TAB (5 TIMES) F4 `print "BACKUP FROM DRIVE 2 TO 1"` ↵ `print "Check cartridge in drive 1"` ↵ `print` ↵

1 Use `edit` to add the following lines to the start of the 'bu' procedure:

```
print "BACKUP FROM DRIVE 2 TO 1"
print "Check cartridge in drive 1"
print
```

Note that the command `print` on its own prints a blank line on the screen, here used simply for cosmetic reasons, to make the display more legible.

When a command finishes in ARCHIVE, the cursor simply returns to a fresh command line. There is no other indication that the job is done, and sometimes ARCHIVE is ready for a new command long before the drives stop working.

↓ ↓ F4 `print "File '"+fn$+"' copied to drive 1"` ↵

2 Add the following line to the end of the procedure to let you know when it has finished:

```
print "File '"+fn$+"' copied to drive 1"
```

Note the single quotation marks inside the double ones. This is the way to print a quotation mark without the error message

`missing closing quote` appearing. The construction, `"'"`, creates a string with a single quote in it.

If the backup command fails for any reason, the procedure will stop and the completion message line will never be reached.

The procedure now looks like this:

```
proc bu
  print "BACKUP FROM DRIVE 2 TO 1"
  print "Check cartridge in drive 1"
  print
  input "Full file name to copy? "; fn$
  backup fn$ as "mdv1_"+fn$
  print "File '"+fn$+"' copied to drive 1"
  endproc
```

↵ ESC 3 Leave insert by pressing **ENTER**, and then **ESC** to leave `edit`.

`save` ↵ `wedfile1` ↵ Type `save` and press **ENTER**, and then type `wedfile1` and press **ENTER** to save the procedures.

`bu` ↵ 5 Type `bu` and press **ENTER**.

`wedfile1_prg` ↵ 6 Type `wedfile1_prg` and press **ENTER**. The follwing procedures now make up the `wedfile1_prg` file:

```
a, b, backupboth, bg, bm, bu, cg, closeboth, cm, d,
e, i, makecopies, n, og, om, openboth, start
```

`new` ↵ 7 Type `new` and press **ENTER** to clear everything from memory.

SAVING AND LOADING PROCEDURES

Loading procedures – the LOAD command

There will be no procedures in memory if you have just started ARCHIVE, or if the memory has been cleared. Before you can use any of the procedures you have just created, you must make a copy of them in temporary memory by reading them off the tape. The command `load` loads a procedure file into memory.

`load` ↵ `wedfiles` ↵

1 Type `load "wedfiles"` and press **ENTER** before issuing any other commands.

`start` ↵ 2 Type `start` and press **ENTER** to open the wedding files.

Alternatively, use the command `run "wedfiles"` to do exactly the same thing as these two separate commands. Use the `makecopies` command periodically to back up your data files.

Note: Long programs may be loaded from the tape in fits and starts, with long pauses before the command finishes. Do not worry as this is normal. Be patient with `load`.

Notes on saving procedures

- If you merge several programs, the procedures cannot be separately re-saved again. Unless you delete procedures from those in memory, every one of them will be saved to the filename you use with the command `save`.
- To keep the programs separate, and manageably small, clear the procedures from memory before starting a new program or re-loading one you want to change.

A procedure to clear procedures but not data

It is often necessary to clear procedures from memory after saving them, so that a new program can be started.

The command `new` closes any data files you have open as well as clearing the procedures. This wastes time while the data files are saved on to tape, and they then need to be reopened.

Avoid this problem by creating a program with a dummy procedure in it. It can then be `load`ed at any time to clear all other procedures from memory.

Note: `load`ing a program file into memory clears all the procedures already there, replacing them with its own.

Create this procedure by first clearing the memory with `new`.

`new` ↵

1 Clear everything from temporary memory by issuing the command `new` and pressing **ENTER**.

`edit` ↵ F3 `dummy` ↵ ESC ESC

2 Create a dummy program file containing an empty procedure:

```
proc dummy
  endproc
```

`save` ↵ `dummy` ↵

3 Use `save` to create a program file called `dummy`.

Using the DUMMY procedure file

`load` ↵ `dummy` ↵

1 When you want to clear only procedures from memory, type in `load "dummy"` and press **ENTER**. This will wipe out all procedures in memory.

`edit` ↵

2 Type `edit` and press **ENTER**.

F3 `d` ↵

3 Before starting on your new procedures, delete `dummy` by pressing **F3**, typing `d`, and pressing **ENTER**.

Notes on saving and loading procedures

- Procedures, like data files, are created and changed only in memory, and have to be saved on tape if you do not want to lose them when the machine is switched off, or the memory cleared.
- Similarly, when you load ARCHIVE, any procedures you want to use must first be loaded from the tape into memory.
- Saving and loading procedures is like closing or opening data files. One command saves them onto the tape, the other reads them back into memory. The difference is that saving procedures, unlike closing data files, still leaves the procedures in memory. It makes a copy of them on the tape, but does not clear them from memory.
- Procedures are also stored on tape in files. These are program files rather than data files, and contain only procedures, not records. Up to 255 procedures may be stored in one file if there is enough room on the tape.
- Procedures are stored on tape using file names. The rules for procedure file names are the same as for data files – they can be up to 8 characters long, starting with a letter, with no spaces. The file name does not have to correspond in any way to the names of the procedures themselves.
- ARCHIVE automatically allocates a file name extension of `_prg`, which stands for `PRoGram`. You can of course supply your own, but must remember to use it every time you load or save the file.
- Procedures are saved onto tape using the command `save`. eg. `save "procfile"`.
- Procedures are loaded from tape into temporary memory using the command `load`. eg. `load "procfile"`. The command `load` wipes out any procedures already in temporary memory and substitutes its own.
- The command `run` loads a program in exactly the same way, but executes the procedure called `start` as soon as it has done so.
- To load a procedure file without wiping out procedures which have already been loaded in memory use the command `merge`. eg. `merge "procfil2"`. If any procedure in the second file has the same name as one already in temporary memory, the *new* one will *replace* the *old* one.

SUMMARY

Using the `count()` function with the `print` command, Miranda is able to see how many names and addresses she still has to type in, and how much posting the invitations will cost.

She makes life easier for herself by creating procedures to move about the file quickly. These procedures have one-letter names, which call ARCHIVE commands like `edit`, `display` and `alter`. These procedures are saved with the file name `easy_prg`.

Miranda then goes on to create further procedures to make opening, closing and backing up of files much quicker and simpler. These procedures she calls `wedfiles_prg`. She calls the `easy` procedures back into temporary memory using the `merge` command, so she can use them at the same time as the `wedfiles` procedures. She creates a procedure called `start` which will be the procedure ARCHIVE first executes if the command `run` is used to load the procedures.

After this, Miranda creates a general purpose backup procedure, using a variable and the `input` command to get the file name to be backed up. Finally, she creates a 'dummy' procedure which will clear all other procedures from temporary memory when it is loaded, without closing any open data files.

Helpful hints

- Keep procedures short.
- Make procedure names as meaningful as possible, so that you remember what they do. Other procedures that use them will then be easy to understand.
- Make variable names as short but as meaningful as possible.
- Keep programs small (only a few procedures) so that they can easily be `merged` to achieve more complex tasks (without having to delete lots or retype any).
- Use `print` statements in procedures to let you know what is going on, especially if they involve waiting an uncertain amount of time.
- Use `rem` commands to remind you what your programs and procedures are for.
- When developing and experimenting with procedures, save different stages of the program onto tape with different file names. This way, if you change your mind about changes, you don't have to change back the changes. Simply `load` an older version of the program which was `saved` before the changes were made.

CHAPTER 5

Loops

One of the greatest benefits of using a computer to handle large amounts of data is its ability to perform tediously repetitive tasks.

ARCHIVE happily does the same thing over and over again, for hour after hour. Typing just a few simple commands once is enough to set ARCHIVE checking or changing every single record in a file.

This chapter introduces loops, the commands which make this possible. You will create a procedure for stepping through every record in a file. You will also write a procedure to copy data from one file into another, effectively allowing data file structures and fields to be changed.

The following features of ARCHIVE are introduced:

- The ARCHIVE commands:

  `all/endall` – repeat a set of commands for all the records in the file
  `while/endwhile` – repeat a set of commands while certain conditions persist
  `position` – make a specific record the current record
  `trace` – switch on/off a line-by-line program trace
  `let` – allocate a value to a variable
  `update` – amend a record, using current field values
  `kill` – delete a file

- The ARCHIVE functions:

  `recnum()` – displays record number
  `eof()` – indicates that a record is the last one in the file

Points to remember

- The loop is the most fundamental, useful and labour-saving device a programming language has to offer.

  The most common causes of loops failing or sticking are:

- starting them in the wrong place
- failing to increment the loop counter/record pointer
- forgetting to tell them when to stop.

Example: Joyce takes over – fast forward!

Joyce suspects Miranda may be removing some of her friends from the list and blaming it on ARCHIVE. To make sure that no dowdy but indispensable dowager has been displaced, or some socially inexcusable mispelling or omission committed, Joyce inspects the list.

As soon as *Top of the Pops* ends and the twins relinquish the television set, she plugs in the QL, retunes the set to the QL's frequency, and loads ARCHIVE.

If you are not just starting ARCHIVE, save any procedures in memory and use `new` to clear all data and procedures from memory.

Starting up

`run "wedfiles"` ↵

1 She puts the data tape in the right-hand drive and enters `run "wedfiles"`.

`display` ↵

2 She types `display`, and the first record of the most recently opened file, `"guests"`, appears on the screen.



Joyce strikes back . . .

A PROCEDURE TO MOVE QUICKLY THROUGH THE FILE

Despite the ease of using Miranda's procedure `n`, or `next`, and F5, Joyce sets about making life even easier for herself by writing a procedure to step through the records automatically.

First she clears the procedures that were automatically loaded when she ran `wedfiles`. She does this by loading the almost empty program file, `dummy`.

Instructions

`load` ↵ `"dummy"` ↵

1 Type `load "dummy"` and press **ENTER**
The single, empty procedure `dummy` is loaded into memory, and all other procedures wiped out.

`edit` ↵ 2 Type `edit` and press **ENTER**.

F3 `d` ↵ 3 Delete the `dummy` procedure by pressing F3 and then `d` followed by **ENTER**
The procedure vanishes, leaving no procedures at all. ARCHIVE immediately offers the opportunity to create a new procedure, as if you had pressed F3 and `n`.

The ALL command

The command `all`, which is always followed by the command `endall`, goes to the first record in a file and moves consecutively from record to record through the whole file. After the last record, it goes back to the first record and stops. Any commands which appear between `all` and `endall` will be carried out with each new record.

Using ALL in a procedure

`zoom` ↵ 1 Type the new procedure name `zoom` and press **ENTER**

`all` ↵ 2 Type `all` and press **ENTER**

`sprint` ↵ 3 Type `sprint` and press **ENTER**
This command forces the record to be printed in the standard `display` layout, since it does not happen automatically when you use procedures.

`endall` ↵ 4 Type `endall` and press **ENTER**

↵ 5 Press **ENTER** again to leave the insert mode

The procedure should look like this:

```
proc zoom
  all
    sprint
    endall
  endproc
```

ESC 6 Press ESC to leave `edit`.

`zoom` ↵ 7 Type `zoom` and press ENTER
Every record in the file appears on the screen in turn, as fast as ARCHIVE and the QL can find and display them. Occasionally, the Microdrive may start, looking for more records.

The sequence of events is this:

- ARCHIVE reads the command `all` for the first time and moves to the first record in the current file. ARCHIVE now expects to see the command `endall` before the end of the procedure.
- The command `sprint` is carried out on the first record.
- On reaching the command `endall`, ARCHIVE moves on one record and goes back in the procedure to the first command following the word `all`.
- The second record is displayed by `sprint`.
- ARCHIVE moves on one record and the procedure loops back to start the `all`, `sprint`, `endall` sequence again.
- This continues until ARCHIVE has moved through all the records in the file. When the last record has been reached, and the command `endall` tries to move on one, the end of the file has been reached. ARCHIVE cancels the `all`/`endall` sequence and proceeds to read the command following `endall` for the first time. In this case, it is `endproc` and the procedure stops.

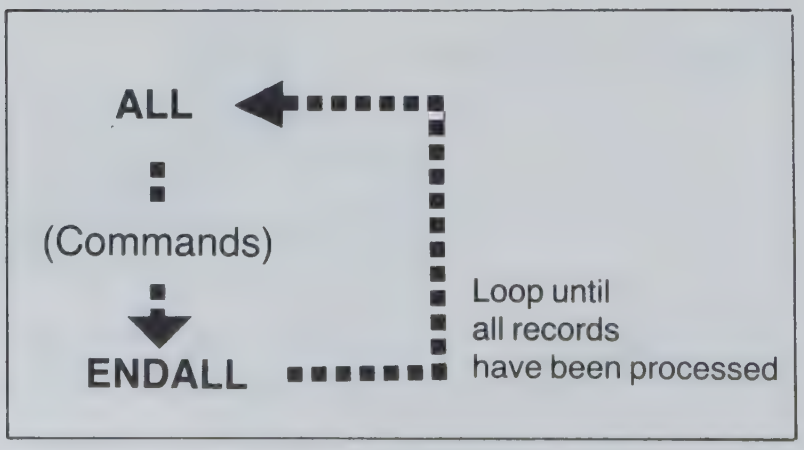


Figure 5.1 – The ALL Loop

Loops

In programming terms, this is called a `LOOP`. Loops repeatedly carry out instructions a specified number of times. In this case, the specified number of times is once for each record in the file.

The command `all` must be followed by the command `endall`, or the procedure will collapse with the error message `bad program structure`.

Similarly, the command `all` cannot be used on its own after the prompt, or ARCHIVE will become hopelessly confused and probably seize up in a very short space of time. You must ensure that if you use `all` in the command line that you include an `endall` after it.

Note: the `all` command is designed to move through a file in the fastest possible way, and usually displays records more or less in the order in which they were created. ARCHIVE uses some short cuts to do this which means that the sequence and content of records it is moving through must not being changed during the course of the loop, eg. by using the `alter` or `delete` commands. The command is primarily for speedy looking only.

Displaying the record number

Joyce's memory not being what is was, she fails to note the absence of the horrid Herberts, so is agreeably surprised by Miranda's work. She still wants to look through her own records again. Unfortunately, the `all` command does just what it says, and moves through all the records. Miranda typed in her mother's list after her own (editing as she went) so Joyce needs to start in the middle, and go on to the end of the file.

She knows that there are 12 records in the file, and that hers begin about half way through.

`position 6` ↵

1 Type `position 6` and press **ENTER**
The first name on Joyce's list appears on the screen. It is the seventh record in the file, which ARCHIVE helpfully numbers as 6, because ARCHIVE counts records starting with 0. Archive displays:

```
Dame Edith Throgmorton-Dunne
```

A procedure to display the record number

`edit` ↵ F3 `nprinrec` ↵ `print"Recordnumber:";recnum()` ↵ ESC

1 Use `edit` to create a new procedure to display the record number:

```
proc prinrec
  print "Record number "; recnum()
  endproc
```

Notice the semi-colon (`;`) separating the text in quotes from the function `recnum()`. The semi-colon tells ARCHIVE to print the two items immediately after each other on the same line. Notice the space after the word `number` which is needed to put a space on the screen between the text and the number.

The RECNUM() function

The function `recnum()` displays the number of the current record. The records are numbered consecutively, more or less in the sequence they were inserted, starting with 0.

Note: ARCHIVE does not always place one record after another in the file in the order they were entered. As records are deleted it may insert others in their place to be economical on space.

ESC 2 Return to the command line.

`prinrec` ↵ 3 Type `prinrec` and press **ENTER**
The line `Record number: 6` should appear at the bottom of the DISPLAY AREA, without clearing the display.

Note: When a `print` command is issued at the command line, any record being displayed is cleared from the DISPLAY AREA. When used in a procedure, however, the screen display stays on and `print` prints everything on the bottom line unless you specify otherwise.

Display and Procedures

The standard display layout in memory behaves in a different way when used with procedures.

Firstly, when the current record changes, the display does not show the new record automatically. You must tell it to by using the command `sprint` in the procedure. This is short for 'screen print', and prints the values of the current field variables on the display screen.

Secondly, the `print` command does not cause the display to be cleared from the screen. Unless you specify otherwise, the results of `print` commands all appear on the bottom line of the screen.

The `sprint` command has no effect if the standard display is not on the screen. Use the `screen` command to re-display the screen if it is already in memory, or the `display` command if it is not.

Using the PRINREC procedure

`first` ↵ 1 Type `first` and press **ENTER**
The screen display shows the first record.

`prinrec` ↵ 2 Type `prinrec` and press **ENTER**
The number on the bottom line should change to 0. In fact, ARCHIVE prints the whole line again, but since none of the other characters change, this is not obvious.

Note: The record numbers are not fixed or absolute. They always stay in sequence, even if the order of the records is changed, or records are deleted. Any record's number can change.

Joyce decides to edit `zoom` so that it starts from the current record, whichever that is, and moves forwards to the end.

The WHILE loop

`edit` ↵ 1 Type `edit` and press **ENTER**

TAB 2 To reach the procedure `zoom` press the **TAB** key
The word `zoom` is highlighted on the left of the screen.

↓ F5 3 Cursor down to the line `all` and press **F5**

CTRL ↓ 4 Press **CTRL** and the ↓ key together to delete the word `all`.

`while not eof()` ↵ 5 Type `while not eof()` and press **ENTER**

↓ ↓ F5 6 Cursor down to the word `endall` and press **F5**

→ → → CTRL ↓ 7 Press the → key to move over the letters `end`. Hold down the **CTRL** key and press ↓ to delete the letters `all`.

`while` ↵ 8 Type `while` and press **ENTER**

↑ F4 9 Cursor up to the line `sprint` and press **F4** to insert a line after it.

`prinrec` ↵ 10 Type `prinrec` and press **ENTER**

The procedure now reads:

```
proc zoom
  while not eof()
    sprint
    prinrec
    endwhile
  endproc
```

The two commands `while` and `endwhile` cause a loop in the procedure similar to the `all`, `endall` loop. The function `eof()` is used to check if the end of the file has been reached. The loop will continue while, or as long as, the `eof()` function returns the value 0. In other words, until the end of the file.

Notice that ARCHIVE indents the commands between `while` and `endwhile` in the same way that all the commands between `proc` and `endproc` are indented. This makes it easier to identify the commands involved in the loop and see the structure of your procedure.

↵ 11 Press **ENTER** to leave insert mode

ESC 12 Press **ESC** to leave `edit`

> ***The EOF() Function***
>
> The function `eof()` is used to detect when the end of a file has been reached.
>
> `eof()` returns the value 1 if the end of the file has been reached, and 0 if it has not.
>
> The command `last` makes the last record in the file the current record, and immediately sets `eof()` to 1.
>
> The command `next` only sets `eof()` to 1 when `next` is issued while already on the last record. In other words, issuing the command `next` while on the second to last record makes the last record the current record, but does not set `eof()` to 1. Issuing the command again does.

Using the amended ZOOM procedure

`zoom` ↵ 1 Type `zoom` and press **ENTER**
Nothing happens. The cursor remains on top of the > prompt, indicating that the command has not finished running. Pressing any key except **ESC** will have no effect.

ESC 2 Press **ESC**
<ESC> appears and the cursor is waiting on a new line.

Obviously, something is wrong with the procedure `zoom`. Either it is moving through the file, and not displaying the change, or it is not moving from the starting record.

TRACING MISTAKES

To watch ARCHIVE running a procedure:

`trace` ↵ 1 Type `trace` and press **ENTER**
Nothing appears to happen. You have simply told ARCHIVE that you want to be able to trace the execution of a procedure.

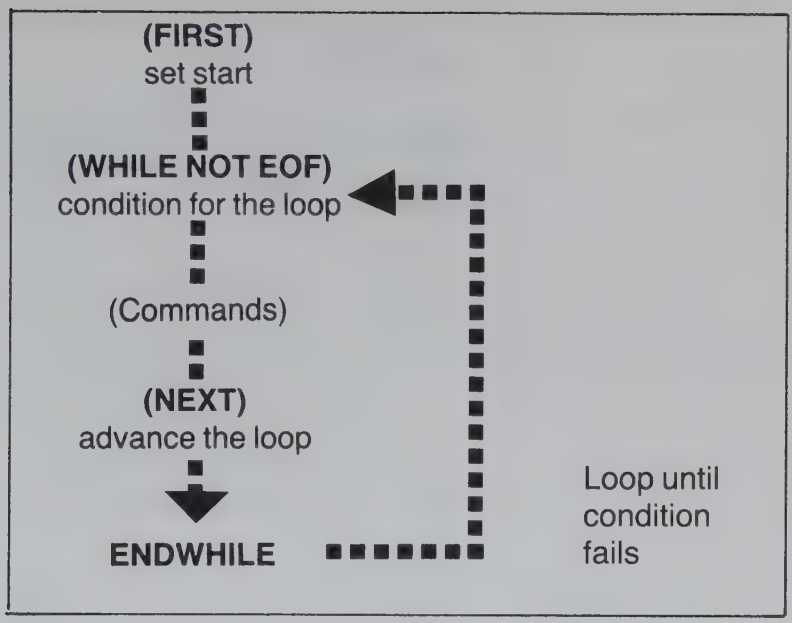


Figure 5.2 – The WHILE NOT EOF Loop

`zoom` ↵ 2 Type `zoom` and press **ENTER**
Each line of each procedure appears in the WORK AREA as ARCHIVE reads it. The command `sprint` takes a little longer than the others, which flash past too fast to read easily.

SPACE (HOLD DOWN)

3 To pause execution of the procedure, press the **SPACE BAR** and hold it down.

SPACE (RELEASE) 4 To let the procedure continue, release the space bar.

SPACE (HOLD DOWN) 5 Use the **SPACE BAR** to watch the procedure in action.

The sequence of events is as follows:

```
zoom      :while not eof()
zoom      :sprint
zoom      :prinrec
prinrec   :print "Record number: ";recnum()
prinrec   :endproc
zoom      :endwhile
zoom      :sprint ...
```

— and so the whole sequence starts again. The only line that does not reappear is the first: `while not eof()`.

Notice how easy it is to see the effect of the second procedure, `prinrec`, being called by `zoom`.

ESC 6 Press **ESC** to cancel the procedure.

As it stands, the procedure will never end. It has caught itself in a 'forever loop'.

It will never reach the end of the file because ARCHIVE receives no commands to move from the record it is on. Unlike the `all` loop, the `while` loop has other uses besides stepping through the records of a file, and must be told to do so.

`edit` ↵ TAB

1 Use `edit` and tab to the `zoom` procedure, in order to change it.

↓ (3 TIMES)

2 Use the ↓ key to get to the line `prinrec`.

F4 `next` ↵

3 Press F4 and insert the line `next`, which tells ARCHIVE to move on to the next record when it has carried out the instructions in the procedure `prinrec`.

The procedure should now look like this:

```
proc zoom
  while not eof()
    sprint
    prinrec
    next
    endwhile
  endproc
```

↵ ESC

4 Press ENTER to leave insert mode, and then press ESC to leave `edit`.

`zoom` ↵

5 Type the newly edited command `zoom` at the > prompt. This time, the display changes to show each record in turn, and its record number is incremented on the bottom line.

You can also see the lines of the procedure flashing past in the WORK AREA, because `trace` is still on.

After the procedure reaches the last record, it stops, and the cursor reappears after the > prompt on the command line.

Turning off TRACE

The procedure ran more slowly than it need have done because the `trace` function was on.

To switch it off:

`trace` ↵

1 Type `trace` and press ENTER
The command `trace` is like an on-off switch: give the command once and `trace` is switched on; give it again and it is switched off. This sort of on-off control is called a *toggle*. Another example of a toggle is the F2 key.

The TRACE command

The `trace` command prints each line of a program or procedure in the WORK AREA as it is executed.

The first time the command is entered, the trace facility is switched on and remains on until the command is entered again.

It is possible to switch `trace` off and on even while a procedure is running, by pressing CTRL and `t` together. This combination of keys acts exactly as if the `trace` command had been given.

Holding down the SPACE BAR while `trace` is on and a procedure is running suspends the execution of the procedure; when the SPACE BAR is released, the procedure continues.

It is useful to leave the trace on while developing procedures. It slows down the speed at which procedures run, however, and should not be needed once programs are working.

`zoom` ↵ 2 Type `zoom` and press ENTER
The cursor returns on a new line and nothing happens. This is because the file is already at the end, and so the `while not eof()` loop never even starts.

`position 6` ↵ 3 Type `position 6` and press ENTER
The seventh record appears in the display, but the printed record number still reads `11`. To change this printed number you must give the `prinrec` command again.

`prinrec` ↵ 4 Type `prinrec` and press ENTER
The bottom line now reads `Record number: 61`. The new record number, 6, has overwritten the first 1 of the 11, but not the last.

Editing PRINREC

`edit` ↵ 1 Call up `edit` in order to make a simple correction to `prinrec`.

↓ 2 Press the down arrow key to get to the line `print`.

F5 ↓ 3 Press F5 to edit the line, and press the down arrow key to get to the end of the line.

`;" "` ↵ 4 Add `;" "` to the end of the line, so that it reads:

```
print "Record number: ";recnum();" "
```

This causes a space to be printed after the number, however long it is. This should ensure that record numbers up to 99 are blanked out when overwritten.

ESC 5 Press ESC to leave edit.

The procedure now looks like this:

```
proc prinrec
  print "Record number: "; recnum(); " "
  endproc
```

zoom ↵ 6 Type zoom and press ENTER
This time, the procedure steps through the records one at a time, printing the record number at the bottom of the screen after doing so. When the end of the file is reached, after record number 11, the procedure stops of its own accord.

Using ZOOM with another file

You can use the same procedure, unchanged, to browse through the memos file.

use "m" ↵ 1 Type use "m" and press ENTER

zoom ↵ 2 Type zoom and press ENTER
The Microdrive may start, looking for the bulk of the "memos" file, and the record number on the bottom of the screen increments from 0 to 14. Apart from that, nothing happens. This is because the display is still set for the "guests" file.

display ↵ 3 Type display and press ENTER to replace the standard display of the "guests" file with that of the now current "memos" file. The current record is the last one, make a present list.

first ↵ 4 Type first and press ENTER

zoom ↵ 5 Type zoom and press ENTER
The records flash past much faster than those in the "guests" file and are not so easy to read. You can easily adapt the loop you have already created to print lines one after another on the screen.

CREATING A PROCEDURE TO LIST THE FILE

The procedure zoom can be adapted to list the file. Before changing zoom:

save ↵ zoom ↵ 1 Save the procedures to a program file called zoom.

This contains the procedures prinrec and zoom, as follows:

```
proc prinrec
  print "Record number: "; recnum(); " "
  endproc
proc zoom
  while not eof()
```

```
    sprint
    prinrec
    next
    endwhile
endproc
```

edit ↵ 2 Type edit and press ENTER

TAB 3 Press TAB to get to the zoom procedure.

F5 CTRL↓ listmemos ↵
4 Change the procedure name to listmemos.

↓ ↓ F5 CTRL↑ ↓ note$ ↵
5 Change the line sprint to read: print note$

↓ F3 c ↵ 6 Cut the line prinrec.

The procedure now reads:

```
proc listmemos
  while not eof()
    print note$
    next
    endwhile
endproc
```

ESC 7 Press ESC to leave edit.

Replicating a procedure

You have changed the procedure zoom into listmemos, but you have not lost zoom. It is saved on tape in the file "zoom_prg". You can restore it without affecting either listmemos or prinrec by using merge.

merge "zoom" ↵ 1 Type merge "zoom" and press ENTER

edit ↵ 2 Now type edit and press ENTER to see the effect of the last command.

Both zoom and listmemos exist. This is because you saved zoom to a tape file before you changed it to listmemos. When the file was merged back in, prinrec was replaced by the incoming, identical prinrec, but neither of the other two procedures was present either in the temporary memory or on tape.

Use this technique to make similar, but slightly different procedures without retyping them in full each time.

By saving procedures individually, you can recombine them in a different way in several different programs, without having to start again each time.

Consider creating a library of procedures on a Microdrive cartridge.

In summary, it is possible to replicate a procedure by saving it, changing its name, then merging it back into memory.

ESC 3 Press ESC to leave `edit`.

Using the listing procedure

first ↵ 1 Type `first` and press ENTER

listmemos ↵ 2 Type `listmemos` and press ENTER
All 14 records flash past on the bottom line of the display, overwriting each other as they appear.

As explained above, PRINT statements in procedures do this when the standard display is on the screen.

edit ↵ F4 cls ESC
3 Use EDIT to insert the command `cls` as the first line after the procedure name in the procedure `listmemos`. This will clear the screen display. You will find that many major procedures need to begin with a command which sets or clears the display.

The procedure now looks like this:

```
proc listmemos
  cls
  while not eof()
    print note$
    next
    endwhile
  endproc
```

ESC first ↵ 4 Leave EDIT and make the first memo the current record.

listmemos ↵ 5 Type `listmemos` and press ENTER
Each memo in the file is printed on a separate line in the DISPLAY AREA.

The result should look like this:

```
see Vicar
ask Felicity to be a bridesmaid
ask Jayne to make their dresses
ask Mrs. Glazer to arrange the flowers
don't forget buttonholes
white shoes
buy a dress
has Nigel got a morning suit ?
book Pickwick the photographers
car
phone Sarah - where did they find their caterers ?
how many people can we pack into the Tudor Room at the Dunkery Hotel ?
order a cake
find suitable Hotel for staying guests
make a present list
```

Printing the list on paper

If you have a printer, all you need to do to produce a list on paper is change the `print` command to `lprint`. The use of printers is more fully discussed in the next chapter. This is what the procedure will look like if you want it to print its output on paper:

```
proc listmemos
  cls
  while not eof()
    lprint note$
    next
    endwhile
  endproc
```

Multiple commands on one line

It is possible to enter several commands on one line, if each is separated by a full colon (`:`).

```
all:print note$:endall ↵
```

1 Type `all:print note$:endall` and press **ENTER**
The result is exactly as if you had given the command `first` followed by `listmemos`. Using this technique allows you to use commands on the command line which need to be followed by a matching `end...` command like `endall` or `endwhile`. The maximum line length, however, is 255 characters, including spaces, and would obviously be inconvenient for most purposes, especially as you cannot save the line.

Typing several commands on one line can also be a useful device in procedures for making them more compact and readable.

Saving the procedures

Now save the existing procedures as `"zoom_prg"`. This contains the following procedures:

```
proc listmemos
  cls
  while not eof()
    print note$
    next
    endwhile
  endproc
proc prinrec
  print "Record number: "; recnum(); " "
  endproc
proc zoom
  while not eof()
    sprint
```

```
        prinrec
        next
        endwhile
    endproc
```

```
kill ↵ zoom_prog ↵
```
1 Remove the old `zoom_prog` from the tape using `kill`.

```
save ↵ zoom ↵
```
2 Type `save "zoom"` and press **ENTER**

```
F5 SHIFT ↓ ↓ mdv1_ ↵
```
3 After checking that your backup data tape is in drive 1, save the procedures to `mdv1_zoom` by editing the line.

ADDING FIELDS TO A FILE

Looking at the memos, Joyce realises that Miranda might as well keep her notes on scraps of paper for all the help that ARCHIVE can give her, with the `"memos"` file as it is. It is not possible to tell easily which tasks need to be done urgently, or whether they involve hair-raisingly fraught shopping expeditions or simply a phone call.

She decides to add two new fields to the `"memos"` file, one for the date tasks should be done by, and another to contain codes indicating what needs to be done.

It is impossible to alter the record structure of the `"memos"` file, so she will have to create a new file and transfer the memo records to it.

Creating the new file structure

If you have forgotten how to use `create`, or if you get into trouble, refer to the beginning sections of Chapters 2 and 3.

```
create "newmemos" logical "n" ↵
```
1 Type `create "newmemos" logical "n"` and press **ENTER**

```
note$ ↵
```
2 Type `note$` and press **ENTER**
This is the same as the original field name from the `"memos"` file.

```
date$ ↵ action$ ↵
```
3 Also type the following two field names:
```
date$
action$
```

```
↵
```
4 Finish with the `create` command by pressing **ENTER** on a blank line.

Appending records from the old file to the new

```
edit ↵
```
1 Type `edit` and press **ENTER**

```
F3 n transfer ↵
```
2 Create a new procedure called `transfer`.

`cls` ↵ 3 The first line, `cls`, clears the screen so that it is easy to see what is happening.

`all "m"` ↵ 4 Type `all "m"` and press **ENTER**
There are no records in the new file, so the `all` refers to the old "memos" file.

`let n.note$=m.note$` ↵
5 Type `let n.note$=m.note$` and press **ENTER**
This command makes the `note$` fields in the new file equal to the `note$` field in the original file.

`append "n"` ↵ 6 Type `append "n"` and press **ENTER**
This command inserts a record in the new file with the value of `note$` as set by the previous line.

This is like using the command `insert` to add records, and then pressing F5 to append that record to the other records in the file. `append`, however, can be used without the need to wait for input from the keyboard.

`endall` ↵ 7 End the loop by typing `endall`.

The procedure should look like this:

```
proc transfer
  cls
  all "m"
    let n.note$=m.note$
    append "n"
    endall
  endproc
```

ESC 8 Leave `edit`.

Note that the logical file name has been used explicitly throughout.

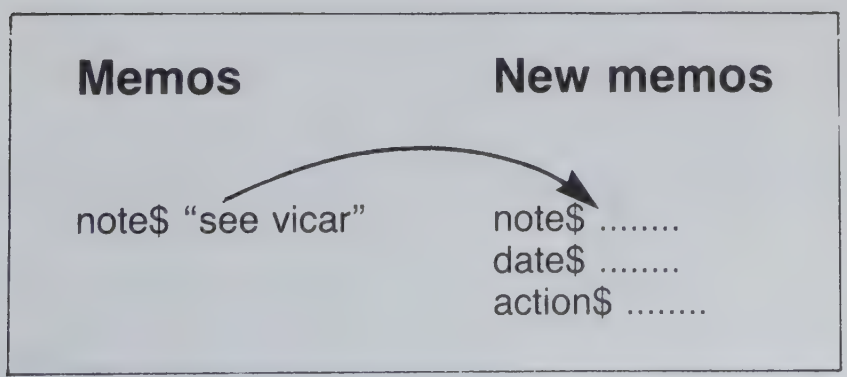


Figure 5.3 – Appending records from the old file to the new file

The LET command

The `let` command gives the field variable on the left of the equals sign the same value (ie. contents) as the field variable on the right. Since both variable names are the same, the logical file name followed by a full stop is used together with the name to distinguish them.

The line can be paraphrased as let the `note$` field in the file `"n"` have the same value as the `note$` field in the file `"m"`." To put it more generally, 'let (ie. make) the first field become equal to the second'.

Polishing the TRANSFER procedure

As it stands, the procedure will work, but you will not know until it is finished that it has done so. Put in some `print` statements to let you watch the progress of the transfer.

↓ (4 TIMES) F4 `prinrec` ↵ `print n.note$` ↵

1 Insert the following lines into the `transfer` procedure after the line `append`:

```
prinrec
print n.note$
```

Note that no logical file name is used in `prinrec`. This should not matter as the first record in the old file should become the first record in the new file.

The procedure should now look like this:

```
proc transfer
  cls
  all "m"
    let n.note$=m.note$
    append "n"
    prinrec
    print n.note$
    endall
  endproc
```

ESC 2 Leave `edit`.

`transfer` ↵ 3 Type `transfer` and press **ENTER**
The screen clears, each new memo is displayed on the screen as it is added to the file. The procedure ends when there are no more records in the old `memos` file.

`display` ↵ 4 Type `display` to see the new file structure. The screen clears and the last record is shown.

The `transfer` procedure has no further use in its present form, and the other procedures in memory have already been saved as `"zoom_prg"`, so close all the open data files and clear the memory by using `new`.

new ↵ 5 Type new and press ENTER.

Renaming a file

There is no single command for renaming a file on the tape. To do this, you must first make a copy of the file you wish to rename, giving the copy file the new name. Delete the file with the wrong name afterwards.

backup "newmemos_dbf" as "memos_dbf" ↵

1 Type backup "newmemos_dbf" as "memos_dbf" and press ENTER
This makes a duplicate copy of the file "newmemos" and, in the process, wipes out the original "memos" file.

F5 ◗ SHIFT◗ CTRL◗ (3 TIMES) SHIFT◗ SHIFT◗ ◗ ◗ mdv1_ ↵

2 Check that your backup cartridge is in drive 1 and then make a back-up copy of the new "memos" file before deleting its original the newmemos file.

Deleting a file with KILL

Before deleting any files, do a directory of the tape to check what is on it.

dir ↵ ↵ 1 Type dir and press ENTER twice
A list of the files on drive 2 appears on the screen.

kill ↵ newmemos_dbf ↵

2 Type kill and press ENTER, then type newmemos_dbf between the quotation marks and press ENTER

dir ↵ ↵ 3 Type dir and press ENTER twice
You can see that this has wiped out the "newmemos" file on drive 2.

The KILL command

Be careful when using the kill command as files cannot be recovered once they have been deleted. If you do not have a back-up, and make a mistake, this can be a disaster.

The kill command needs a full file name with extension.

Note: If you use an incomplete file name, or one that does not exist, ARCHIVE does *not* let you know that it has not deleted anything. Check the directory using dir if in doubt.

- ARCHIVE will refuse to kill an open data file

The kill command usually returns control to the command line quickly, even though the drives keep whirring.

Changing all records

Now that the new file is safely backed up, information needs to be added to the new, blank fields.

Miranda is planning a late June wedding, dreaming of roses and strawberries, and long, light days. Joyce thinks of her own wedding, when it poured with rain.

All those people, and so much to do! She decides to put today's date, April the 5th, on everything.

Instead of typing it all by hand, she wonders how to get ARCHIVE to put the date in every record and a 'p' action code (for phone). She can change the few exceptions by hand.

The obvious candidate for the task is an `all` loop, but the special way this command has been designed to operate means that it cannot be used when records are being changed in length in any way. The `first, while not eof (), next, endwhile` sequence must be used instead.

- ***Remember:*** Do not use the `all` loop on a file while affecting its records with commands like `alter, insert, delete, append` or `update`.

Updating records

The command `update` works in a very similar fashion to `append`, but has the effect of changing an existing record, like `alter`. The fields are given new values using the `let` command, but do not actually replace those in the current record until the command `update` is used.

Create a procedure called `memoupdate` which uses the `update` command.

`edit` ↵ F3 `n memoupdate` ↵

1 Use `edit` to create a new procedure called `memoupdate`.

`first` ↵

2 Type `first` and press **ENTER**
to make the loop start at the first record.

`while not eof()` ↵

3 Type `while not eof()` and press **ENTER**
This loop will make ARCHIVE carry out the following commands on every record until it reaches the end of the file.

`let date$="04/05"` ↵

4 Type `let date$="04/05"` and press **ENTER**
This sets the value of the `date$` field to "04/05" (today's date – the 5th of April).

Notice that the date has been entered in the American fashion, with the month first. This is because ARCHIVE can be used to

Field variables and temporary memory

The commands `update` and `append` take advantage of the fact that the field variables can exist independently of the record containing those fields.

Whenever a record becomes the current record, the contents of its field variables are stored in memory under the relevant field names. In this way the commands `alter` and `insert` allow you to put information in without affecting the record until you save the variables to it, by pressing **F5**.

The commands `update` and `append` act like the **F5** key, and save the contents of any field variables to the record which matches the file structure. `update` affects the current record, and `append` creates a new record.

The field variables can be given new contents, or assigned new values, using commands like `let` and `input`.

The field variables of every opened file remain in memory until changed by commands like `let`, `alter` and `insert`, or until cleared by `close` or `new`. This is why the display continues to show the last record of a file when all the records have just been deleted.

put the records into date sequence if it is entered this way round.

Note: The slash '/' in the date between the day and the month is just to make it more readable, it serves no function in ARCHIVE. If you do use the slash to separate the day, the month and the year, make sure you always use it, otherwise ARCHIVE will not be able to sort properly on the date.

`let action$="p"` ↵

5 Now set the value of the `action` field to `p`.

`update` ↵ 6 Use the command `update` to give the `date$` and `action$` fields of the current record the values set above.

`next` ↵ 7 Type `next` and press **ENTER**

`endwhile` ↵ 8 Finish off the `while` loop by typing `endwhile` and press **ENTER**

The procedure should look like this:

```
proc memoupdate
  first
  while not eof()
    let date$="04/05"
```

```
    let action$="p"
    update
    next
    endwhile
endproc
```

↵ ESC 9 Press ENTER to leave insert mode, and press ESC to leave `edit`.

Note: Be careful when using `update` and `append`, as both commands replace the contents of the fields with the current values of the field variables. For example, if you `append` records onto the end of existing records, and use `let` to assign values to only some of the fields, the other fields in every appended record will take the value of those in the current record at the time appending started. `update` updates every field in the record with the values for the current record in memory.

Time and date functions

Time and date play an important role in many programs, from diaries and project planning, to accounting and time-costing.

ARCHIVE provides several useful functions to let you know the date and time, or manipulate date strings.

`month()` returns the name of a month when supplied with a number. If the number is higher than 12, ARCHIVE divides by 12 and uses the remainder, eg. `print month(15)` displays `March`.

`time()` returns the time as a string representing hours minutes and seconds. You must first have set the QL's clock using the SUPERBASIC command `adate` or `sdate`.

`date()` returns the QL's *system* date (set with the SUPERBASIC command `sdate`) in one of three forms.

For example, if you had to set the system date to the 5th of April 1984, it could be represented as follows:

`print date(0)` displays `1984/04/05`
`print date(1)` displays `05/04/1984` (English Fashion – day first)
`print date(2)` displays `04/05/1984` (American Fashion – month first)

`days()` can be used to calculate the number of actual days elapsed between two dates. This can be very useful in accounting and financial programs.

ARCHIVE and dates

As far as ARCHIVE is concerned, the date is just a string of characters, and 04/05 (April 5) comes before 05/01 (May 1), which is as it should be.

Stored in the American fashion, 05/01 (1 May) comes after 04/05 (5 April), as if they were the numbers 501 and 405. Stored in the English fashion, 01/05 (1 May) comes before 05/04 (5 April) whereas 06/05 (6 May) comes after – as if they were the numbers 105, 504 and 605.

Adding the data

```
open "memos" logical "m" ↵ first ↵ memoupdate ↵
```

1 Open the `memos` file, starting at the first record, and run the procedure `memoupdate`.

```
display ↵
```

2 There are no `sprint` or `print` commands in it, so you will only know it has functioned correctly by typing `display` and pressing **ENTER** when the Microdrives stop whirring.

The `memos` file should now look like this:

```
Logical : m
note$   :  see Vicar
date$   :  04/05
action$ :  p
```

```
edit ↵ F3 d ↵ ESC
```

3 Now delete the procedure `memoupdate`, as it is not going to be used again.

Altering each record in the file

Joyce now types a set of commands in a loop which will offer her the chance to `alter` every single record in the file. She chooses `while` for the loop, as she will be making changes.

`alter` only affects one record at a time, so she can stay as long as she likes on any record and move onto the next by finishing with `alter`.

Note the use of `sprint`, since the automatic display does not work with procedures, and each successive record may appear to have the contents of the previous one if you do not force a screen print.

Type the command as a single line after the > prompt, divided by full colons (:). Do not press **ENTER** until you are finished.

```
first:while not eof():sprint:alter:next:endwhile ↵
```

1 Type `first:while not eof():sprint:alter:next:endwhile` and press **ENTER**

If you make a mistake typing the line, and ARCHIVE responds

with an error message, remember that you can edit the line by first pressing F5.

Do not press ESC while in `alter` unless you want the whole loop to stop.

When using Version 1, press F5 to move on to the next record.

When using Version 2, press F4 to move on without making changes, or press F5 or ENTER, to save changes and move on to the next record.

Note: In `alter`, if you have ARCHIVE version 1, press F5 to move on to the next record, whether you have made any changes or not.

2 Change the action code and date for the following records:

```
Memo                              Date        Action
see Vicar                         04/08
don't forget buttonholes          06/23
white shoes                       04/14       Shop
buy a dress                       04/14       Shop
has Nigel got a morning suit ?    04/05
make a present list               04/05
```

`insert` ↵ `don't forget needle and thread` ↵ `06/23` ↵ F5 (OR F4)

3 Add the following note to the file:

`don't forget needle and thread`

```
note$                                                    date$ action$
see Vicar                                                04/08
ask Felicity to be a bridesmaid                          04/05 p
ask Jayne to make their dresses                          04/05 p
ask Mrs. Glazer to arrange the flowers                   04/05 p
don't forget buttonholes                                 06/23
white shoes                                              04/14 s
buy a dress                                              04/14 s
has Nigel got a morning suit ?                           04/05
book Pickwick the photographers                          04/05 p
car                                                      04/05 p
phone Sarah - where did they find their caterers ?       04/05 p
how many people can we pack into the Tudor Room at the
Dunkery Hotel ?                                          04/05 p
order a cake                                             04/05 p
find suitable Hotel for staying guests                   04/05 p
make a present list                                      04/05 p
don't forget needle and thread                           06/23
```

```
close ↵
```
4 Now close the memos file.

```
backup "memos_dbf" as "mdv1_memos_dbf" ↵
```
5 Make a backup copy of the memos file onto the backup tape in drive 1.

SUMMARY

Joyce decides to write a procedure which will help her look at files easily and quickly on the screen. This procedure, called zoom, uses the sprint command to display records on the screen. It uses a loop, which tells ARCHIVE to repeat a given set of commands, in this case until it reaches the end of the file (using the while not eof() loop).

Joyce also writes a procedure which displays the record number of the current record on the screen. This procedure is called prinrec. It uses the function recnum().

Joyce decides to make the memos file more useful, by adding two more fields to it. To do this she first has to create a new file structure with the new fields in it (action$ and date$), as well as the original field (note$). She then writes a procedure to transfer all the note$ fields from the original file to the new file. She updates the date$ field in the new file by writing a procedure, memoupdate, which uses the commands let and update.

Helpful hints

- Check that the word endproc is indented 2 spaces before finishing a procedure with a all or while command in it. If it isn't, you have forgotten an endall or endwhile.
- Insert the line endwhile immediately after the line while when writing complex loops in procedures. That way it won't be forgotten, and the indentation of the lines after while stays correct. Start to insert all commands involved in the loop by cursoring up and pressing F4.
- Make copies of your procedures in memory by saving them, changing the procedure names, then merging back the saved file.
- Tidy up your tapes occasionally by deleting old and unwanted files, otherwise the tape directory will become cluttered and you may have difficulty finding your way around your own files. Microdrives work less well as they become full.
- Use a notebook to keep details of what each file does. You will forget very quickly.
- To turn off the command trace when a procedure is running, type CTRL and t.

CHAPTER 6

Listing and Printing Information

Some tasks, such as annotating lists, are performed better on paper than on computer. Paper print-outs also provide a last-ditch security against the unthinkable happening – losing all your data on tape.

This chapter explains the printing of lists and labels on the printer as well as the screen.

The following ARCHIVE features are introduced:

- The ARCHIVE commands:

  `lprint` – sends information to the printer
  `paper` – sets the background screen colour
  `ink` – sets the screen character colour

- The ARCHIVE functions:

  `numfld()` – returns the number of fields in a record
  `fieldv()` – returns the contents of a numbered field
  `val()` – converts a string to a numeric value

Points to remember

- Don't use a command which sends information to the printer if you don't have one. The computer will 'hang' or 'lock' if you do.
- If you do want to use a printer, make sure that it is properly 'installed' first. If it isn't, the computer might lock up – at best, garbage will be printed.
- Misuse of the final semi-colon in `print` commands is one of the commonest causes of wrongly formatted output on the screen or printer.

Example: Printing lists

A pile of jewellery by the computer alerts Miranda to the fact that Joyce has been up to her usual tricks. To check her mother's additions to the `memos` file, she lists everything in `memos` on the screen.

A PROCEDURE TO LIST THE MEMOS FILE

Instructions

If ARCHIVE is not loaded, load it now (see Chapter 1), and put your data cartridge with the `guests` and `memos` files on it into drive 2. This cartridge should also contain the programs `zoom`, `easy` and `wedfiles`.

`open "memos" logical "m"` ↵

1 Open the `memos` file with logical name `m`

`load "zoom"` ↵ 2 Type `load "zoom"` and press **ENTER** to load the procedures in the `zoom` program.

`edit` ↵ 2 Type `edit` and press **ENTER**

F4 `first` ↵ 3 Press **F4**
This inserts the line `first` at the top of the procedure `listmemos`.

↓ (3 TIMES) F5 ↓ `;" ";date$;" ";action$` ↵
4 Add the other fields by typing `;" ";date$;" ";action$` at the end of the line `print note$`.

Notice that the `print` command is followed by a list of items, each one separated by a semicolon. The three fields are separated by literal spaces. Without these spaces the fields would all run together in one long, illegible string.

The procedure should now look like this:

```
proc listmemos
  first
  cls
  while not eof()
    print note$; " "; date$; " "; action$
    next
    endwhile
  endproc
```

ESC 5 Press **ESC** to leave EDIT

`listmemos` ↵ 6 Type `listmemos` and press **ENTER**
The whole file lists line by line on the screen, scrolling up out of

the DISPLAY AREA as soon as the list reaches the bottom of the screen:

```
see Vicar 04/08
ask Felicity to be a bridesmaid 04/05 p
ask Jayne to make their dresses 04/05 p
ask Mrs. Glazer to arrange the flowers 04/05 p
don't forget buttonholes 06/23
white shoes 04/14 s
buy a dress 04/14 s
has Nigel got a morning suit ? 04/05
book Pickwick the photographers 04/05 p
car 04/05 p
phone Sarah - where did they find their caterers ? 04/05 p
how many people can we pack into the Tudor Room at the Dunkery Hotel ? 04/05 p
order a cake  04/05 p
find suitable Hotel for staying guests 04/05 p
make a present list 04/05
don't forget needle and thread 06/23
```

The bottom line is blank, and the cursor is sitting at the first position of the new line.

Carriage returns

To fit more information on the screen, Miranda changes the procedure to stop the display from moving on to a new line each time a new record is listed.

`edit` ↵ 1 Edit the procedure `listmemos`.

↓ (4 TIMES) F5 ↓ ; ↵

2 Add a semicolon to the end of the `print` line, so that the line reads:

```
print note$;" ";date$;" ";action$;
```

ESC 3 Press ESC to leave EDIT

The procedure should now look like this:

```
proc listmemos
  first
  cls
  while not eof()
    print note$; " "; date$; " "; action$;
    next
    endwhile
  endproc
```

```
listmemos ↵
```

4 Type `listmemos` and press **ENTER**
This time the records are run together one after the other, wrapping to fill each line of the display until they finish:

```
see Vicar 04/08  ask Felicity to be a bridesmaid 04/05 pask Jayne to make
their dresses 04/05 pask Mrs. Glazer to arrange the flowers 04/05 pdon't
forget buttonholes 06/23  white shoes 04/14 sbuy a dress 04/14 shas Nigel
got a morning suit ? 04/05 book Pickwick the photographers 04/05 pcar
04/05 pphone Sarah -- where did they find their caterers ? 04/05 phow many
people can we pack into the Tudor Room at the Dunkery Hotel ? 04/05 porder
acake  04/05 pfind suitable Hotel for staying guests 04/05 pmake a present
list 04/05 don't forget needle and thread 06/23
```

The cursor is positioned immediately after the last character printed. This is the effect of putting a semicolon at the end of a print statement or print list – new lines are not started after each print command.

Carriage returns

The carriage return is named after the action of old mechanical typewriters, which returned the paper carriage physically to the start position of a new line whenever the return lever was pressed. On electric typewriters, this is done by pressing a large 'J' shaped key exactly like the **ENTER** key on the QL.

The **ENTER** key is often called the **RETURN** key for this reason. It still has the effect of ending a line of text in most programs.

A carriage return is actually two separate actions. One feeds the paper (in a typewriter or printer) up one line, and is known as a 'line feed', often abbreviated as **LF**. The other returns the print-head or carriage to the left margin, and is known as a 'carriage return', or **CR**.

The phrase 'carriage return' is often loosely used to describe the effect of both these actions. The combination starts a new line. The `print` command actually sends an **LF** and a **CR** together as a 'carriage return' unless it is suppressed by ending the statement with a semi-colon.

Final semi-colons and the PRINT command

A `print` command without a semicolon at the end starts a new line when it is finished. It sends a signal to the screen to move the cursor down one line and return to the left margin. This is called a 'carriage return'.

A `print` command on its own sends a carriage return to the screen. This has the effect of producing a blank line.

The carriage return is suppressed when a print statement ends with a semicolon. The command line `print;` would thus have no effect whatsoever.

Printing always starts at the cursor position.

Tabs

Miranda realises that her last idea was not exactly a good one, and sets about rectifying her mistake and enhancing the look of the listings.

The procedure `listmemos` as it stands produces a mess. Even without the semicolon the list was not very clear, since the dates and codes do not align, and are not easy to distinguish, and the `action$` code runs into the following `note$`.

`edit` ↵ 1 Edit the procedure `listmemos`

2 Change the print line by replacing the literal spaces with tab clauses, so that the line reads:

```
print note$;tab 55;date$;tab 61;action$
```

↵ 3 Press **ENTER** to save the new line

The procedure should now look like this:

```
proc listmemos
  first
  cls
  while not eof()
    print note$; tab 55; date$; tab 61; action$
    next
    endwhile
  endproc
```

ESC 4 Press **ESC** to leave EDIT

`listmemos` ↵ 5 Type `listmemos` and press **ENTER**
The following should appear on the screen with the date and code fields neatly aligned in columns:

```
see Vicar                                                    04/08
ask Felicity to be a bridesmaid                              04/05 p
ask Jayne to make their dresses                              04/05 p
ask Mrs. Glazer to arrange the flowers                       04/05 p
don't forget buttonholes                                     06/23
white shoes                                                  04/14 s
buy a dress                                                  04/14 s
has Nigel got a morning suit ?                               04/05
book Pickwick the photographers                              04/05 p
car                                                          04/05 p
phone Sarah - where did they find their caterers             04/05 p
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?                                                      04/05 p
order a cake                                                 04/05 p
find suitable Hotel for staying guests                       04/05 p
make a present list                                          04/05
don't forget needle and thread                               06/23
```

Tabs

Tabulation on a typewriter or computer is the action of making the cursor or print head jump to a specified column before printing any more characters.

The word *tabulate* comes from the Latin word for table – columns of tabulated figures are often referred to as tables.

The ARCHIVE key word `tab` is not a command. It is called a print item, and can only be used together with the commands `print`, `lprint` and `input`.

Tabs jump to an absolute position on the screen, not one relative to the last printed character, eg. `print name$;tab 50` moves to column 50, not a column 50 spaces after the end of the contents of `name$`.

If the specified column is already behind the cursor or print head when `tab` is used, it has no effect.

For example, the statement `print "Today's date: ";tab 5;date$"` printed on a new line would display `Today's date04/03`. After printing the first item, the cursor would already be beyond column 5.

`tab` must be followed by a number no higher than the screen width currently in use.

Notice that the `tab` print item tabulates to the specified column on the screen, and positions the start of the next item printed in that column.

Note: The figures 55 and 61 are used on the assumption that you have the screen display width set to 64 or 80 columns at least.

If one of the memos is longer than 64 characters, it wraps onto the next line. The date and code still appear in the right positions, because the column numbers are higher than the cursor column at the time the `tab` is used.

ADDING COLOUR – NIGEL GETS FANCY

The PAPER and INK commands

Nigel is much taken with Miranda's tabulations, but is less impressed by what he calls 'the brutal mechanical tyranny' of white letters on a black screen.

He sets about exercising his aesthetic judgement on ARCHIVE's colour scheme using the commands `paper` and `ink`.

Try these commands out even if you have a black and white TV. They affect the tones of the screen output as well as the colour.

`paper 4` ↵

1 Type `paper 4` and press **ENTER**
At this stage, nothing happens. The command has instructed ARCHIVE to print the colour 4 (green) as the background colour of any further `print` statements.

`listmemos` ↵

2 Type `listmemos` and press **ENTER**
The DISPLAY AREA instantly turns green and all the memos are listed in white. This is rather illegible, so Nigel decides to change the ink colour.

`ink 0` ↵

3 Type `ink 0` and press **ENTER**
This command instructs ARCHIVE to print any subsequent characters with the colour 0 (black).

`listmemos` ↵

4 Type `listmemos` and press **ENTER**
All the memos are listed in black letters on green.

Miranda is doubtful if spiders on baize is an improvement on brutal mechanical tyranny, but Nigel already has a better idea. How glorious it will be if each note is printed a different colour according to month!

`paper 0:ink 6` ↵

5 Type `paper 0:ink 6` and press **ENTER** to return the screen to its normal colours, white on black.

The PAPER and INK commands

The `paper` and `ink` key words are unusual in being used both as commands in their own right, and as print items with the `print` command.

The `paper` command sets the background colour for all the following text to the colour represented by the value of the following number.

The `ink` command sets the colour of the printed characters (the 'foreground' colour) for all subsequent text to the colour specified by the value of the following number.

Colours:	0 and 1	black
	2 and 3	red
	4 and 5	green
	6 and 7	white

It is possible to use a number higher than 7 to change colours. ARCHIVE first divides the number by 8, and uses any remainder as the number it is interested in.

For example, `paper(2)` is evaluated like this:

```
2/8 = 0 remainder 2 = 2 = red
```

and `paper(16)` like this:

```
16/8 = 2 remainder 0 = 0 = black
```

This is called 'modulo' arithmetic. 'Mod 8' means that the significant value of a number is the remainder after the number has been divided by 8.

Don't use the same colour for both ink and paper, as the result will be a blank screen.

Note: The `paper` command has a slightly different effect depending on where the next print statement starts.

If the cursor is on the bottom of the screen, all subsequent `print` commands cause the full width of the line to have the background paper colour.

If the cursor is anywhere else, only the width of the text actually printed will have the background colour.

To avoid this, use the command `cls` to immediately set the whole screen to the paper colour.

Print items

Nigel plans to use the number of the month to determine the ink colour.

He can do this by incorporating the `ink` command as a print item in a `print` statement. Unlike `ink` used as a command in its own right, colours set in a print item only cover the following items in the `print` statement. As soon as the command is finished, the colours previously in force are restored.

For example, the following four commands print first a red (colour 2), then a white name (colour 6):

```
paper 0
ink 6
print ink 2;"Fred"
print "Jack"
```

All Nigel has to do now is work out how to make the colour of each line change according to the month.

Print items

`print` statements may be comprised of the `print` command followed by a list of items divided by semicolons. These items may include:

- `tab`
- `at`
- `paper`
- `ink`
- information to be printed such as string or numeric expressions

These items can also be used in `input` statements.
Several prompts and variables can be used in a list with a single `input` command. For example, the following command fetches your output for the variables `date$` and `time$` one after the other:

```
input "Date ? ";date$;"Time ? ";time$
```

TAB can be used with `lprint`, but not the other print items.

Using VAL() to get numbers from strings

Nigel's major problem is that the command `ink` needs to be followed by a number or numeric value, and the months are stored as a string of characters in the field `date$`.

He decides to use the function `val()` to convert the month string to a numeric value.

If a string begins with numbers, the function `val()` returns the numeric value of those numbers.

For example, the command `print val("06/23")` will display the number 6.

Notice that `val()` only converts the first number in this string, ignoring the / and subsequent characters, whether numbers or not.

The function VAL()

The function `val()` converts character strings to a single numeric value.

- It returns the value 0 for a string with only text, (ie. no numbers):

  `val("hello")` returns the value 0

- It returns the value of the numbers in the string for a string which starts with numbers:

  `val("123")` returns the value 123
  `val("123hello")` returns the value 123

- It accepts leading spaces, full stops and minus signs, as well as the numbers from 0 to 9, as valid numeric characters:

  `val("  -23.45")` returns the value -23.45

- As soon as `val()` comes across a non-numeric character, it ignores that and any further numbers or characters:

  `val("352 + 345")` returns the value 352

Nigel's second problem is that the numbers of the months April to June are 4 , 5 and 6, *one* apart in value, while the different colours are *two* apart.

Left as they stand, June will produce white letters, but both April and May will produce green letters.

The solution is to multiply the month number by 2. The resulting values will be 8, 10, and 12, which poses a further problem, as these will convert to colours 0, 2 and 4 (remember that numbers higher than 7 can represent colours – to assess which, divide by 8 and the remainder determines the colour). The colour 8 (or 0) is black, the same as the background, and April notes will be invisible. To overcome this, he simply adds 2 to the calculation.

`edit` ↵ 1 Edit the procedure `listmemos`.

↓ (4 TIMES) F5 SHIFT ↓ `ink val(date$)*2+2;` ↵

2 Change the `print` line by inserting the `ink` print item `ink val(date$) * 2 + 2;` after the word `print`, so that the line reads:

```
print ink val(date$)*2+2;note$;tab 55;date$;tab
61;action$
```

ESC 3 Press ESC to leave EDIT

The procedure should now look like this:

```
proc listmemos
  first
  cls
  while not eof()
    print ink val(date$)*2+2; note$; tab 55; date$; tab 61; action$
    next
    endwhile
  endproc
```

`listmemos` ↵ 4 Type `listmemos` and press ENTER
All the memos are listed in the DISPLAY AREA, the April notes being red, and the June notes white.

Nigel wishes there were some notes for May, to add some green to a screen dreary with red.

`save "zoom"` 5 Type `save "zoom"`
This will save the three procedures, `listmemos`, `prinrec` and `zoom` onto tape.

PRINTING ON A PRINTER

The LPRINT command

Miranda wants to study a paper copy of the list, so that all the family can look at it and suggest their own candidates and comments.

`edit` ↵ 1 Edit the procedure `listmemos`.

F5 CTRL↓ (4 TIMES) `print` ↵

2 Change the name of the procedure `listmemos` to `printmemos`.

↓ ↓ F3 `c` ↵ 3 Delete the line `cls`.

↓ ↓ F5 `l` 4 Add the letter `l` to the front of the word `print`

The command `lprint` is almost exactly the same as the command `print` except that it sends its output to a printer instead of the screen.

The `l` on the front of the word probably derives from the term 'line printer', a type of printer capable of printing a full line at a time – widely used with larger computers.

Note: Do not use `lprint` or any other command which sends information to a printer if you do not have one. The QL will 'hang' or jam, as it will be unable to carry out ARCHIVE's command. You will then have to reset the machine *(See 'On Line', below).*

SHIFT↓ CTRL↓ (18 TIMES) ↵

5 Further change the `print` line by removing the `ink` item, as colour print items cannot be used with the printer. The line should look like this:

```
lprint note$; tab 55; date$; tab 61; action$
```

↵ 6 Press ENTER to save the new line.

The procedure now looks like this:

```
proc printmemos
  first
  while not eof()
    lprint note$; tab 55; date$; tab 61; action$
    next
    endwhile
  endproc
```

ESC 7 Press ESC to leave `edit`.

`merge "zoom"` ↵ 8 `merge` the program `zoom` to restore `listmemos` alongside `printmemos`.

`save` ↵ `lists` ↵ 9 Save the procedures to a program file called `lists`

`save` ↵ `mdv1_lists` ↵

10 Save the procedures onto drive 1 as well, as `mdv1_lists`, after making sure your backup tape is in drive 1.

The program `lists` contains the following procedures:

```
proc listmemos
  first
  cls
  while not eof()
    print ink val(date$)*2+2; note$; tab 55; date$; tab 61; action$
    next
    endwhile
  endproc
proc prinrec
  print "Record number "; recnum(); " "
  endproc
proc printmemos
  first
  while not eof()
    lprint note$; tab 55; date$; tab 61; action$
    next
    endwhile
  endproc
proc zoom
  while not eof()
    sprint
    prinrec
    next
    endwhile
  endproc
```

On line

Make sure the printer is ready before you use a printer command. It must be switched on, connected to the QL, and *on line*.

A printer is said to be on line when it is ready to receive information and instructions from the computer, rather than from its own switches. On many printers this is indicated by a light marked SEL or ON LINE. If the printer is not ready, the QL will 'hang' until it is.

Installing printers

Further, the file printer_dat must be on the ARCHIVE cartridge, and it must contain accurate information about your printer. This lets ARCHIVE knows which particular signals (if any) it needs to receive to work properly.

If your printer prints rubbish when sent data by ARCHIVE, this file probably needs correcting. The process of making sure the program knows about your printer is called 'printer installation' and is described in an appendix of the ARCHIVE manual, and in the Blueprint books QL QUILL, QL EASEL and QL ABACUS.

You need the SUPERBASIC program and data files `install_bas` and `install_dat` before you can install a printer.

If they are not on your ARCHIVE program cartridge, run the installation using QUILL, EASEL or ABACUS cartridges which do have them.

Copy the resulting `printer_dat` file onto your ARCHIVE cartridge using `backup`.

Remember to unprotect your program cartridge by covering the notch if necessary.

The `printer_dat` file

The installation program stores information that you give it about your printer in a file call `printer_dat` on the program cartridge. This information is loaded for ARCHIVE's use when ARCHIVE itself is loaded.

This file must always be called `printer_dat`, or ARCHIVE does not recognize it. There is no way of easily telling which printer it is set up for.

Solve this problem by creating a dummy file to act as a label (like the '`_lab`' files used in the backup routines), and save it onto the program cartridge. If you have an *Epson MX-80* printer, for example, it could be called '`mx80_pri`'. You can now use the command `dir` to find out what `printer_dat` is set for.

If you ever need to send printer output to more than one printer, or for more than one page setting, or even to another computer, you will have to install `printer_dat` again.

Avoid having to do this all over again every time by making several `printer_dat` files to start with, one for each requirement. Copy them with different names (eg. `mx80_dat` or `apricot_dat`) onto your data tape in drive 2.

When you need to change the printer, copy the relevant `_dat` file back onto the program cartridge and reload ARCHIVE, eg:
`backup "mx80_dat" as "mdv1_printer_dat"`

Don't forget to relabel the cartridge by deleting the old `_pri` file and saving a new one, so that you know in future which set up `printer_dat` is installed for.

Printing the memos

`printmemos` ⏎

1 Type `printmemos` and press **ENTER**
The printer now lists every note in the memos file, line by line, and stops as soon as it has finished the last line.

Form feeds

Remove the paper by winding it out by hand, or by pressing the printer's *form feed* button, usually marked `FF` or `TOF`. This will probably only work if the printer is 'off line', ie. ready to receive signals directly from its own switches, instead of from the computer.

A form feed 'feeds' the paper through to where the printer thinks the top of the next sheet, (or form) begins.

It would be useful if the computer prevented this manual intervention by sending an instruction to the printer to eject the current page, ie. to do a form feed.

Use `edit` to create the following procedure:

```
proc ff
  lprint chr(12);
  endproc
```

The CHR() function

The function `chr()` is used here to send a code to the printer, which interprets it as a signal to form feed. Note the semicolon at the end to suppress the carriage return which otherwise would also be sent.

Exactly the same code can be sent to the TV and most monitors, using the `print` command and the `chr()` function. It clears the screen, as if the command `cls` had been used. This is the equivalent, on the screen, of starting a fresh page.

The number inside the brackets is called an ASCII code. You don't have to know these codes or understand how they work to use the two or three that are occasionally needed. When they appear it will usually be in a print statement, and always using the `chr()` function, as here. For more information on ASCII and `chr()` see the Appendix at the back of the book.

ESC 1 Press ESC to leave insert mode

TAB (3 TIMES) 2 Press TAB to get to the procedure `printmemos`

↓ (5 TIMES) F4 `ff` ↵

3 Insert a line `ff` after the line `endwhile` to go to the top of the next page

ESC 4 Press ESC to leave insert mode

TAB 5 Press TAB to get to `zoom`
This is already in the program file `zoom`, and is not needed in `lists`.

F3 d ↵ 6 Delete the procedure `zoom` by using F3 followed by `d` and then ENTER

SHIFT TAB F3 d ↵

7 Also delete the procedure `prinrec`

The remaining procedures should be as follows:

```
proc ff
  lprint chr(12);
  endproc
proc listmemos
  first
  cls
  while not eof()
    print ink val(date$)*2+2; note$; tab 55; date$; tab 61; action$
    next
    endwhile
  endproc
proc printmemos
  first
  while not eof()
    lprint note$; tab 55; date$; tab 61; action$
    next
    endwhile
  ff
  endproc
```

ESC save ↵ lists2 ↵ save ↵ mdv1_lists2 ↵

8 Leave `edit` and save these procedures to `lists2`, and to `mdv1_lists2`

LISTING ADDRESSES

Changing the listing procedure to list the guests

The `guests` file can be listed and printed in exactly the same way, the only difference being that it has many more fields than `memos`.

edit ↵ TAB 1 Use `edit` and press the TAB key to call up the `listmemos` procedure.

F5 SHIFT ↓ CTRL ↓ (5 TIMES) guests

2 Change the name of the procedure from `listmemos` to `list-guests`

↵ ↓ (4 TIMES) F5 CTRL↓ print name$ ↵ F4 print address1$ ↵ print address2$ ↵ print address3$ ↵ print address4$ ↵

3 Replace the print line with the following lines:

```
print name$
print address1$
print address2$
print address3$
print address4$
```

print ↵

4 Type print and press ENTER. This prints a blank line between each address, making them easier to read.

↵

5 Press ENTER to leave insert mode

The procedure should look like this:

```
proc listguests
  first
  cls
  while not eof()
    print name$
    print address1$
    print address2$
    print address3$
    print address4$
    print
    next
    endwhile
  endproc
```

ESC

6 Leave edit.

merge ↵ lists2 ↵

7 Merge the program file lists2 to restore the procedure listmemos.

save ↵ lists3 ↵

8 Save the procedures as lists3. This will contain the following procedures:

```
ff
listguests
listmemos
printmemos
```

open ↵ guests" logical "g ↵

9 Open the guests file with logical name g

listguests ↵

10 Type listguests and press ENTER

The selected fields of each record scroll past. Some have more spaces between them than others as the fourth line of the address is sometimes empty.

```
John and Sue Heath
43b Clarkson Road
London
SW16

Sally Jones and Susan Clark
3b Academy Mews
Chelsea
London
SW3
```

and so on.

Passing parameters

It would be tedious to have to change the `print` commands to `lprint` (or back again) for every field of every file that needs listing or printing.

Miranda writes a procedure containing the command so that it only needs changing once.

`edit` ↵ F3 `n line;l$` ↵ `print l$` ↵ ESC

1 Create the following procedure:

```
proc line; l$
  print l$
  endproc
```

ESC 2 Leave `edit`
Notice the semicolon followed by a variable after the procedure name. This acts rather like the `input` command used in Chapter 4 in the `f (find)` procedure. It takes a different value (in this case a string of characters) each time the procedure is used, and stores it as a variable for use in the procedure.

Unlike the `input` command, ARCHIVE does not wait for you to type anything in.

The information must be passed to the procedure at the time you use it as a command. The information passed is called a *parameter*.

Note that as it stands the procedure `line` only works if passed a string expression, since the variable name '`l`' is followed by a $ sign.

Using parameters

`line` ↵ 1 Type `line` and press **ENTER**
ARCHIVE displays the first line of the procedure and an error message `Error 2: end of statement expected.`

If a procedure expects to be passed a value, you must supply one. ARCHIVE thinks that the semicolon and variable name should not

be there (and that the `proc` statement should have ended) because you did not pass on a string variable as a parameter.

```
line;name$ ↵
```

2 Type `line;name$` and press **ENTER**
The name$ field of the current record is displayed:

```
Mrs Agnes Blake
```

Parameters

Pieces of information passed on to a procedure are called *parameters*. They give the procedure information it needs to fulfil its tasks. They make it possible for a single procedure to be used in many different ways by different programs and with different data. This saves having to write new ones all the time, and saves space in programs.

The number of parameters passed to a single procedure is determined by the receiving procedure, where they follow the procedure name. They may be string or numeric variables, and are separated by commas.

The data passed on to the procedure must match in quantity and type (string or numeric) the number of parameters expected.

For example, a procedure starting like this:

```
proc example;char$,num,text$
```

could be used like this:

```
example;"Rabbit",6,breed$
```

All variables set up as parameters on a `proc` line are cleared from memory as soon as that procedure ends. They are called *local* variables, because they are 'local' to that procedure.

It is important to distinguish between the value passed as a parameter, and the variable used to store it. In programming jargon, the variable in the procedure that needs a parameter is called a 'formal' parameter (it is part of the 'form' of the procedure), and the actual data or value passed by the calling procedure is called an 'actual' parameter.

```
line;"Name "+name$ ↵
```

3 Type `line;"Name "+name$` and press **ENTER**
Note that the strings were added together and not divided by a semicolon.

If a procedure expects to be passed a single string, and you attempt to pass more, an error will result.

The two strings are printed on the screen:

```
Name Mrs Agnes Blake
```

edit ↵ TAB TAB ◆ (4 TIMES) F5 SHIFT↑ CTRL◆ line; ↵ ◆ F5 SHIFT↑ CTRL◆ line; ↵ (5 TIMES) ◆ F5 CTRL◆ line;"" ↵ ESC

4 Now use `edit` to change the `listguests` procedure to look like this:

```
proc listguests
  first
  cls
  while not eof()
    line; name$
    line; address1$
    line; address2$
    line; address3$
    line; address4$
    line; ""
    next
    endwhile
  endproc
```

Notice that the two sets of quotation marks in `line;""` are side by side, with nothing at all between them. This is known as an empty string, or in denser jargon yet, a null string.

It is being used here to send something at least to the procedure `line` which must be passed a string parameter, but in this case is required to print only a carriage return.

The null string

A null string, "", is a string which contains nothing at all.

A null string is quite distinct from, for example, the string " ", which has nothing in it as far as the display shows, but to ARCHIVE contains a space, which is a character in its own right.

The null string has an ASCII value of 0, while a space has an ASCII value of 32.

The most common use of the null string is for clearing or declaring string variables so that they do not accidentally have an effect on a program by holding an unwanted value.

It is also the value of an `input` variable if nothing is typed and **ENTER** is pressed.

Saving the procedures

save ↵ lists4 ↵ 1 Save the procedures to `lists4`. This contains the following procedures: `f, line, listguests, listmemos, printmemos`.

edit ↵ TAB ↓ F5 l ↵ ESC 2 To change the whole procedure from printing on the screen to printing on the printer, change the command `print` in the procedure `line` to `lprint`:

```
proc line; l$
  lprint l$
  endproc
```

Editing procedures – cut and paste

Miranda realizes that the procedures `listguests` and `listmemos` are essentially the same apart from the fields printed for the different files.

She decides to cut the `line;` lines from `listguests` and place them in a separate procedure.

Cut

edit ↵ 1 Type `edit` and press **ENTER**

TAB TAB F5 SHIFT ↓ CTRL ↓ (6 TIMES) ↵ 2 Change the name of the procedure `listguests` to `list`

↓ ↓ ↓ 3 Make the `while` line the current, highlighted line.

F4 dispguest ↵ 4 Insert the line `dispguest` to call the new procedure.

↓ 5 Make the line `line;name$` the current, highlighted line.

F3 c 6 Press **F3** and the letter `c`
Read the prompts box. It tells you to use the cursor keys. You only need to do this if you want to cut more than one line, as here.

↓ ↓ ↓ ↓ ↓ 7 Press the down arrow 5 times so that a block of lines down to and including `line;" "` is white. If you go too far, cursor back up.

↵ 8 Press **ENTER**
The whole block of lines disappears and the remaining lines close up:

```
proc list
  first
  cls
  while not eof()
    dispguest
    next
```

```
    endwhile
endproc
```

When lines are cut from a procedure they do not disappear forever immediately. They are stored in an area of memory called a BUFFER.

They remain there until replaced by the next set of cut lines, until you quit `edit` or until transferred somewhere else by a 'paste' operation.

You are going to 'paste' them into a new procedure. Do this immediately in case anything accidentally clears the cut buffer.

Buffers

Buffers are areas of memory which store information sequentially (usually) and then release it in exactly the same order (usually) when required. (This is called a FIFO or 'first in, first out' buffer.)

Most printers have some memory in them, called a printer buffer, which stores information sent by the computer until the printer is ready for it. This is why printers often go on printing long after the computer has finished sending data. Printers usually print much more slowly than computers send.

Most computer keyboards (including the QL) have keyboard buffers, although usually only a few characters can be stored in them at a time. This is why you can usually start typing in commands while the last command is still running, and before the command prompt has reappeared. When it does, the keystrokes stored in the buffer suddenly appear.

ARCHIVE's 'cut and paste' buffer in `edit` works in a similar way. Information is stored in it whenever lines are cut, and then retrieved in exactly the same order when the paste command is used. The main difference is that each new cut operation clears the current buffer *before* the new cut lines are put in the buffer. If you want to move three separate pieces of a procedure, you should:

CUT PASTE CUT PASTE CUT PASTE

and *not*:

CUT CUT CUT PASTE

Paste

F3 n 1 Press F3 and n to start a new procedure.

dispguest ↵ 2 Call it dispguest
The proc line is now the current, white line.

ESC 3 Press ESC to leave insert mode

F3 p 4 Press F3 and p. The last block of cut lines instantly appears in the procedure:

```
proc dispguest
  line; name$
  line; address1$
  line; address2$
  line; address3$
  line; address4$
  line; ""
  endproc
```

The paste process always inserts previously cut lines immediately after the current line, and removes them from the buffer as it does so. Once used, the paste command cannot be used again until after more lines have been cut.

It is possible to move lines within a single procedure, or from one to another.

ESC 5 Leave edit

merge "lists4" ↵ 6 Merge the program file lists4

save ↵ lists5 ↵ 7 Save the procedures to lists5:
dispguest, ff, line, list, listguests, listmemos, printmemos,

trace ↵ 8 Switch on the trace facility if it is not on already, to watch the three procedures, line, list and dispguest in action.

list ↵ 9 Type list and press ENTER to see the result.

```
John and Sue Heath
43b Clarkson Road
London
SW16

Sally Jones and Susan Clark
3b Academy Mews
Chelsea
London
SW3
```

and so on.

trace ↵ 10 Switch trace off again.

Using the FIELDV() function

You can now use the `list` and `line` procedures to display any file you create, item by item, as long as you write a procedure which determines which fields to use.

Miranda thinks this could prove troublesome in the long run. She doesn't fancy the idea of rewriting the listing procedures with every new file she creates.

She writes a procedure to list a record from any file, without even knowing the names of the fields.

`print fieldv(0)` ↵

1 Type `print fieldv(0)` and press **ENTER**
ARCHIVE displays the contents of the first field of the current record:
`Mrs Agnes Blake`

F5 SHIFT ↓ SHIFT ↓ ↓ CTRL ↓ 1 ↵

2 Press F5 and change the 0 to a 1
The second field is displayed:

`Cherry Cottage`

- The function `fieldv()` returns the contents of the field whose number appears between the brackets. The fields are numbered starting at 0 for the first one. The last (eighth) field in the `guests` file is therefore number 7.

A STANDARD LISTING PROCEDURE

`edit` F3 `n disprec` ↵ `let count=0` ↵ `while count<8` ↵ `print fieldv(count)` ↵ `let count=count+1` ↵ `endwhile` ↵ ESC

1 Create the following procedure:

```
proc disprec
  let count=0
  while count<8
    print fieldv(count)
    let count=count+1
    endwhile
  endproc
```

Unlike loops you have encountered so far, this one is not related to moving through records to determine when it should stop. It will be performed a fixed number of times, in this case 8.

The word `count` is a numeric variable, and is being used as a counter which counts the number of times the loop is performed.

When `count` becomes equal to 8, the loop will stop. ARCHIVE can not confuse it with the function `count()`, because it has no brackets.

Remember that numeric variables are identified by not having a dollar sign at the end, and are used to store numbers which are going to be used for calculations.

You cannot store character strings in a numeric variable. Commands like `let count="0"` or `let count=name$` will cause errors.

ESC 2 Leave `edit`

`trace` ↵ 3 Use `trace` to watch the procedure:

`disprec` ↵ 4 Type `disprec` and press **ENTER**

The current record is listed field by field, and line by line:

```
Mrs Agnes Blake
Cherry Cottage
Far Lane
Linton
Cumbria

1
```

`trace` ↵ 5 Turn `trace` off again.

How the procedure works

This is what happens, step by step:

`let count=0`:
When the procedure starts, the command `let` assigns the value on the right of the equals sign, 0, to the variable name on the left. The value 0 is now stored in memory under the name `count`. It could, of course, have been called `counter`, or `x`, or any valid variable name.

This line is the equivalent of the `first` command in `while not eof()` loops. If you fail to zero the counter before a count loop, it may already have a high value, and the loop will be passed over immediately. If the `count` variable does not yet exist, the whole procedure collapses anyway when `count` is used with the `while` command.

`while count<8`:
The first time ARCHIVE reads this line, the value of `count` is 0. The < symbol means 'is less than', and the line can be paraphrased as 'While (the value of) count is less than 8, (do the following commands up to the command `endwhile`)'.

`print fieldv(count):`
In this line ARCHIVE substitutes the value of `count`, 0, in the brackets, and prints the contents of field 0, that is, the first field.

`let count=count+1:`
This line adds one to the value of `count`. ARCHIVE first calculates the value of the numbers on the right of the equals sign. It looks up `count` and substitutes a 0, then adds 1. It stores the result, 1, in memory under the name `count`, replacing the old value.

This line is the equivalent of the `next` line in `while not eof()` loops. It advances the loop one step nearer its end. If you fail to increment the counter, the loop will never end.

`endwhile:`
On reaching the `endwhile` command, ARCHIVE returns to the top of the loop and checks to see if the loop should continue.

The value of count is now 1, which is less than 8, so the loop continues. ARCHIVE starts again with the first command after `while`.

When the value of `count` reaches 8, the `while` test governing the loop will fail and the result is that the program will jump past the loop. The command immediately after `endwhile` will be executed.

Note that although the loop is not performed when the counter reaches 8, eight passes of the loop have been performed, because the count started at 0.

Using the standard listing procedure

Miranda tries out this new achievement on the `memos` file.

`edit` [TAB] [TAB] [TAB] [TAB] ▼ (4 TIMES) [F5] [CTRL] ▼ `disprec` ↵ [ESC]

1 Use `edit` to change the `list` procedure so that it calls the new general purpose procedure `disprec` instead of `dispguest`.

The procedure should now look like this:

```
proc list
  first
  cls
  while not eof()
    disprec
    next
    endwhile
  endproc
```

`use "m"` ↵ 2 Use the `memos` file.

list ↵ 3 Type `list` and press **ENTER**
The contents of each of the three fields in each record is displayed, followed by sets of irrelevant numbers. ARCHIVE is trying to find a fourth and more fields which don't exist, and is dredging up other data from memory: .

```
see Vicar
04/08


7.32443419E-34
6.97575925E+19
0
5.17554062E-20
ask Felicity to be a bridesmaid
04/05
p

1.19352688E-20
4.29568997E+24
4.79410114E-20
4.31428399E+24
```

and so on.

The function NUMFLD()

Fortunately, Miranda's all-purpose listing procedure will not now be useless for all files with less than eight fields.

The function `numfld()` returns *the number of fields in a file* (the current file, unless a logical filename appears between the brackets).

print numfld() ↵ 1 Type `print numfld()` and press **ENTER**
The number 3 appears on the screen.

edit ↵ TAB ↓ ↓ F5 ↓ CTRL ↑ numfld() ↵ ESC
2 Change the 8 in the procedure `disprec` to `numfld()`, so that the line now reads:

```
while count<numfld()
```

The procedure should now look like this:

```
proc disprec
  let count=0
  while count<numfld()
    print fieldv(count)
    let count=count+1
    endwhile
  endproc
```

The procedure `list` will now work with any file.

`list` ↵ 3 Type `list` and press **ENTER** to see this working properly with the `memos` file:

```
see Vicar
04/08

ask Felicity to be a bridesmaid
04/05
p
ask Jayne to make their dresses
04/05
p
```

and so on.

`save` ↵ `lists6` ↵ 4 Save the procedures to `lists6`.
This contains the procedures: `dispguest, disprec, ff, line, list, listguests, listmemos, printmemos,`

Printing out procedures with LLIST

The command `llist` prints out the contents of every procedure in memory on the printer. Note the double `l` at the start of the word. The first `l` has the same provenance as the `l` of `lprint`, and the 'list' part derives from the expression 'program listing', which is what `llist` produces.

The procedures appear indented and in alphabetical order, as they do in `edit`.

Paper listings of procedures can be easier to work with when using a complicated set of procedures. They can also be annotated and used for following through the procedure step by step to find mistakes and *bugs*. (Otherwise known as *dry-running* a procedure.)

Paper procedure listings can be used to analyse the program step-by-step, making a note of variable values involved in each line.

The figure (6.1) shows how the error in the previous version of `disprec` might have been traced by 'dry running'.

```
proc
  let count=0
  while count<8
    print fieldv(count)
    let count=count+1
    endwhile
  endproc
```

```
Times through Loop

    1              2              3              4

    0
    0<8            1<8            2<8            3<8
    0=note$        1=date$        2=action$      3 = ???
    0+1=1          1+1=2          2+1=3

                                                 no field no. 4
```

Figure 6.1 – Sample Dry Run

A *bug* is computer jargon for anything that causes a program to fail or behave in a way it isn't meant to. The word is usually reserved for mistakes that are hard to find, or which crop up only occasionally under unexpected circumstances.

Make sure the printer is ready before using `llist`.

Deleting outdated program files

The program `lists` has gone through several stages, being updated each time, and saved to a new version number. There are now 7 versions: `lists`,`lists1`,`lists2`,`lists3`,`lists4`, `lists5` and `lists6`. `lists6` is the most up-to-date, and contains all the relevant procedures from the other versions.

```
kill↵ lists_prg↵ kill↵ lists1_prg↵ kill↵ lists2_prg↵ kill↵
lists3_prg↵ kill↵ lists4_prg↵ kill↵ lists4_prg↵ kill↵ lists5_prg↵
```

1 Delete all the versions of `lists` except `lists6` to make more room on the tape.

```
backup↵ lists6_prg" as "lists_prg"↵ kill↵ lists6_prg↵
```

2 Rename `lists6` to `lists`. To do this, first you have to use the `backup` command to copy `lists6` to the filename `lists`, then you must delete `lists6`.

```
dir↵ ↵
```

3 Do a directory of drive 2 to check that you now have the following program files on tape: `easy_prg`, `wedfiles_prg`, `dummy_prg`, and `lists_prg`.

Check the date on the `_lab` file in drive 2

```
dir↵ mdv1_↵
```

4 Do a directory of your backup tape, in drive 1, to check the date on the `_lab` file to make sure it is the correct backup tape. If it is the same as your working tape, use your other backup tape.

`new` ↵ 5 Clear all procedures and data files from temporary memory by typing `new` and pressing **ENTER**.

`backup ...` 6 Backup files created or changed since the tape was last used.

`save "Apr6_lab"` ↵ F5 SHIFT ↓ ↓ `mdv1_` ↵
7 Save label files with today's date onto both the working tape and the backup tape in drive 1.

8 `kill` the old `_lab` files on both tapes.

See Chapter 12 for more information on listing data, including label printing.

SUMMARY

In this chapter, Miranda uses different commands and procedures to print the information from her two files `memos` and `guests` both on the screen and on the printer.

First, she creates a procedure called `listmemos` which lists each field of the `memos` file on the screen, line by line. She uses a semi-colon at the end of a `print` line to stop a Carriage Return being printed. She then uses the `tab` command to make each field in the file print at a specified place on the screen.

Nigel feels that Miranda's lists need livening up a little, so he uses the `paper` and `ink` commands to add colour to them. He uses the function `val()` to change the colour of each memo according to its date.

Now Miranda creates a procedure to list her memos on the printer so she can see them on paper. She calls this procedure `printmemos`, and she changes the `print` command that she used in the `listmemos` procedure to `lprint`. This sends the information to the printer rather than to the screen. She uses the `chr()` function in a procedure called `ff` which she creates to make the printer do a 'form feed' at the end of the page.

After this, Miranda turns her attentions to the `guests` file, and she changes the `listmemos` procedure so that it will work with this file. Then she creates a new procedure called `line` which will mean she doesn't have to change every single `print` command to `lprint` in the procedure if she wants to print her `guests` file on the printer rather than the screen. To do this, she passes the procedure 'parameters'. These enable one procedure to be used in many different ways.

Next, Miranda uses the cut and paste facility in the procedure editor to create a new procedure called `dispguest`, which lists the `guests` file on the screen. She uses the `fieldv()` function to enable the `list` procedure to be used with any file. To do this, she creates a count loop, which adds 1 each time it prints a field, and

when it gets to the end of the record, returns the counter to 0. She also uses the function `numfld()` so that files with different numbers of fields can use the procedure.

Finally, Miranda prints out a list of all her procedures using the `llist` command. She then deletes the various versions of the program `lists` which have now been superseded by the final version, `lists6`.

Helpful hints

- If nothing happens when you send a print command to a properly installed printer, you have probably forgotten to press the printer's on-line switch.
- Make sure there is a date on your printed file and program listings. Otherwise, when you have printed the same file more than once, you may forget which list is the latest.

CHAPTER 7

Selecting Information

There is no point storing lots of information on a computer unless it can be retrieved whenever needed. ARCHIVE retrieves data much faster and more accurately than any manual or paper filing system. Searching for records by any of their contents is fundamental to using a database, and can be done in a variety of ways.

In this chapter and the next, you will learn the methods available and the pros and cons of using different commands to search for and select your data, and write a procedure to make using `continue` easier.

You will also learn about methods for checking data entry (such as dates) using logical expressions.

The following ARCHIVE features are introduced:

- The ARCHIVE commands:

  `search` – finds a record by a specific field
  `select` – selects records which match certain criteria
  `reset` – returns file to normal

- The ARCHIVE functions:

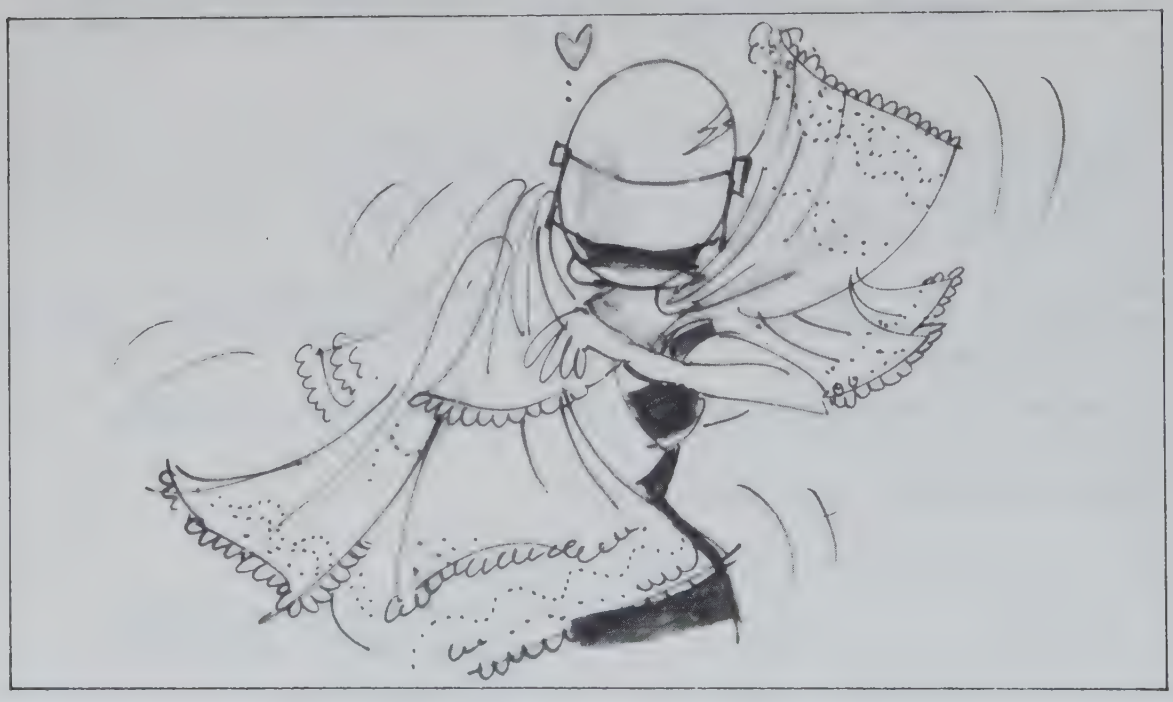
  `upper()` – converts letters to upper case
  `lower()` – converts letters to lower case
  `found()` – checks if a `FIND` has been successful
  `instr()` – finds certain characters anywhere in a string of characters
  `len()` – returns the length of a string as a number representing the number of characters in the string

- The operators – `>`, `<` (greater than and less than), `and`, `or` and `not`.

- String slicing

Points to remember

- `continue` continues with the last `search` or `find` command, no matter how long ago it was used, starting after the current record.
- Use `reset` when finished with a `selection`, or subsequent commands and searches will be wrong.
- ARCHIVE behaves as if `unselected` records simply do not exist (temporarily). Commands like `all` only affect (all) the selected records.



A Tight Fit

Example: Wedding preparations – what and when?

Starting up

If you do not have ARCHIVE loaded, load it now. You should not have any data files open.

Put a backup cartridge in Microdrive 1, and use `dir` to check that the `_lab` file (see chapter 3) is not the same as that on the working data cartridge.

FINDING SPECIFIC RECORDS

Instructions

look↵memos" logical "m↵

1 Type look and press ENTER, then type memos" logical "m and press ENTER to open the memos file.

Note that ARCHIVE only provides the first set of quotation marks automatically if you press ENTER when you need them. To avoid cursoring over one to type the rest of the line, simply do not type one after the logical name.

display↵ 2 Type display and press ENTER

load↵ zoom↵ 3 Load the program file zoom

Miranda wants to see all the memos that remind her to make phone calls.

The command find, introduced in Chapter 2, is of no use in finding the notes coded with p in the action field, as every record with a p anywhere in any text in any field, including note$ itself, will be found if you type the command find "p", eg. Pickwick the photographers

The SEARCH command

The command search finds records by comparing specific fields with specific words or values. Like the find command, it starts with the first record in the file and checks it to see if it matches in the requested way. If it doesn't, search checks each successive record until it finds one that does. If a match is found, the search can be continued with the continue command. continue saves the last record which matches as the current record if no more are found.

search action$="p"↵

1 Type search action$="p" and press ENTER
The display changes to show the second record:

ask Felicity to be a bridesmaid

continue↵ 2 Type continue and press ENTER
This will display the next record where action$ = p:

ask Jayne to make their dresses

F5↵ 3 Press F5 followed by ENTER to see more
Only records with the letter p at the start of the action$ field are shown, until there are no more. The following records should be selected:

```
ask Mrs. Glazer to arrange the flowers
book Pickwick the photographers
car
phone Sarah -- where did they find their caterers ?
how many people can we pack into the Tudor Room at the Dunkery Hotel ?
order a cake
find suitable Hotel for staying guests
```

The `search` command is absolutely pedantic about what it looks for, both in checking the case of letters and the length of the line. For example, the command `search action$="p"` will ignore the record if the field contains the two characters 'p ' (ie. the letter p followed by a literal space).

The command `search` is unlike `find` in that it takes notice of whether a letter is in upper or lower case. The command just used would not have found records with the letter `P` in the `action$` field. The match is said to be *case specific*.

The functions UPPER() and LOWER()

If you are in any doubt about whether data is stored in upper or lower case, force the characters into upper or lower case using the functions `upper()` or `lower()`. The function `upper()` converts lower case letters to upper case, without affecting any other characters (eg. numbers). The function `lower()` does the reverse.

For example:
`search upper(action$)="P"` will find both upper and lower case ps. Try it to make sure.

This does not permanently change the contents of the fields, it merely means that when it comes to check for a match, ARCHIVE converts any lower case letters to upper case before comparing them with the letters being searched for.

A PROCEDURE TO KEEP FINDING SPECIFIED RECORDS

Deliberate forever loops

Miranda wants to write a procedure to save having to use `continue` at the command line each time she wants to see the next record which matches.

`edit` ↵ F3 `n seemore` ↵ `while 1=1` ↵ `continue` ↵ `sprint` ↵ `endwhile` ↵ ESC

1 Use `edit` to create the procedure `seemore`, and type it in as follows:

```
proc seemore
  while 1=1
    continue
    sprint
    endwhile
  endproc
```

This use of the `while` command causes a loop that will never stop, as the result of the equation will always be true. The value 1 will always be equal to the value 1.

The only way to break a 'forever' loop while it is running is to press **ESC**.

ESC 2 Leave `edit`

```
save↵ seemore↵
```

3 Save the procedure to `seemore`.

Matching an empty field with a null string

Before using the new procedure, Miranda starts another search. She now wants to check general memos, (those unrelated to phoning or shopping, and stored with nothing in the `action$` field.)

```
search action$=""↵
```

1 Type `search action$=""` and press **ENTER**
Note the use of a null string. Remember that it is different from " ", which contains a space, a character in its own right. A null string has the ASCII value 0, while a space has the ASCII value 32.

A null string only works with character or text strings and fields. Empty numeric fields contain the value 0. They cannot contain nothing at all.

The first record in the file appears:

```
see Vicar
```

`seemore↵` 2 Type `seemore` and press **ENTER**
The four remaining records which match the search criterion appear in turn.

```
has Nigel got a morning suit ?
don't forget buttonholes
make a present list
don't forget needle and thread
```

ESC 3 Press **ESC** to cancel the forever loop.

The IF command

It is obviously inconvenient and clumsy to leave `seemore` as it stands.

`edit` TAB TAB ↓ ↓ F4 `if found()=0` ↵

1 Insert these lines into the procedure:

```
if found()=0
```

This line checks to see if the function `found()` returns a 0 (no find) or a 1 (record found). The line could also have been written `if not found()`.

`stop` ↵ 2 `stop`
The command `stop` causes all procedures to stop, and returns control to the command line.

`endif` ↵ 3 `endif`
Like `while ... endwhile`, the command `if` must be followed by an `endif`, so that ARCHIVE knows which commands are related to the `if`.

The procedure should now look like this:

```
proc seemore
  while 1=1
    continue
    if found()=0
      stop
      endif
    sprint
    endwhile
  endproc
```

↵ ESC 4 Press **ENTER** to leave insert mode, and **ESC** to leave `edit`.

`save` ↵ `seemore` ↵ F5 SHIFT ↓ ↓ `mdv1_` ↵

5 Save the procedure to the program `seemore`, and, after checking that you have your backup tape in drive 1, save it as `mdv1_seemore`, using **F5** to save retyping.

Two new commands, `stop`, and `if` followed by `endif`, have been introduced in this alteration, along with a new function, `found()`.

The `if` clause here may be paraphrased as: 'If (matching record) not found, stop'.

The function FOUND()

The function `found()` is similar to the `eof()` function in that it returns a value of 1 if it is true, and 0 if it is not. That is, if the `search` or `continue` commands find a match, the value of `found()` is 1, or true. Otherwise, if no match is found, `found()` will be equal to 0.

The function `found()` works with the command `find` in exactly the same way.

In version 2 of ARCHIVE, `found()` also works with `locate` (described in the next chapter).

IF and ENDIF

The construction `if .. endif` is similar to `while .. endwhile` in that the commands are paired and must not exist without each other. Also, any commands that come between them will only be executed if the equation or function that follows the first command is true.

For example, `while not eof()` performs commands in the loop while it is true that the end of the file has not been reached.

The command(s) between `if` and `endif` in this example are performed when it is true that `found()=0`, that is, if no match has been found. Until then, the command in the `if` clause is ignored and the procedure acts on the first line after `endif`.

True and false

Commands like `if` and `while` determine whether the expression following them is true or false according to whether it returns a 1 or a 0.

Comparisons and matches (using operators like =), conditional tests (`if` and `while`), and states (like end of file) all rely on the idea of true and false. Records match and commands are performed if the relevant expression is true, and not if it is false.

Computers represent the outcome of these evaluations as either 1 or 0. Concepts such as right and wrong, true and false, success and failure, yes and no are reduced simply to non-zero (1), and zero (0) values.

This explains why, for example, the ARCHIVE manual says that the syntax of the `search` command requires a numeric expression (ie. `search(n.exp)`). This does not mean that `search` only matches numeric fields, but that it requires an expression to evaluate (or return) a numeric value representing true or false (match or no match).

The notion of an expression being true or false is central to the way that the commands `if` and `while` work. The commands evaluate the expressions that follow them, and execute the commands up to `endif` and `endwhile` only if the result is 1, or true. If the result is 0, or false, `if` commands are ignored, and a `while` loop ends.

Evaluating Expressions

An expression is any combination of values, whether strings (text), numbers or functions, which can be evaluated to give a single result.

The term value covers both the numeric value of a number or numeric field, and the contents of a string.

The following are examples of expressions:

```
16+1
count()/50*.16
```

These two examples are numeric expressions, and contain no strings (the function `COUNT()` returns a number).

`name$+" Esq."` evaluates as `"John Wooten Esq."`
`date$+" "+upper(action$)` evaluates as `"04/03 P"`
`action$="p"` evaluates as *true* if they are equal

These three examples are string expressions, and contain no numbers unless they are part of a character string.

Evaluating Relative and Logical Expressions

Note that if the last example evaluates as true it returns the number 1. If the result were false, it would return the number 0.

The command `print 10=13-3` displays the number 1, showing that the expression evaluates as correct, or true.

The command `print 10=2*3` displays the number 0. The expression is wrong, or false.

The command `let x=name$="Nigel"` at first looks like a mistake, as it appears to mismatch numeric and string expressions. In fact, the expression `name$="Nigel"` evaluates as 1 if the field contains `Nigel`, and `x` takes the value of the result.

Note the different functions here of the symbol '='. The first one is part of the `let` command, and assigns the value following it to the variable before it. The second is a relative operator in a string expression, and is used to return a true or false value. This is then assigned to `x`.

Using the SEEMORE procedure

`first` ↵ 1 Use `first` to go to the start of the memos file then

`screen` ↵ 2 use `screen` to display the first record.

`seemore` ↵ 3 Type `seemore` and press **ENTER**
Despite the fact that no `search` command has just been given, the procedure shows all the records that match the last search command (`search action$=""`), and stops when it can find no more. It should select the following records:

```
see Vicar
has Nigel got a morning suit ?
don't forget buttonholes
make a present list
don't forget needle and thread
```

The command `continue` in `seemore` will act on the last `search` or `find` command, no matter how long ago it was issued, as long as the memory has not been cleared. `continue` starts its search on the first record after the current record, whether that matches or not.

Greater than and less than

`search action$>""` ↵

1 Type `search action$>""` and press **ENTER**
The first record where `action$` has a `p` in it appears:

```
ask Felicity to be a bridesmaid
```

`seemore` ↵ 2 Use `seemore` to see all the matches.
The symbol > stands for 'greater than', and ARCHIVE has matched all the memos with `p` and `s`, that is, `action$` fields which have more than nothing in them:

```
ask Jayne to make their dresses
ask Mrs. Glazer to arrange the flowers
white shoes
buy a dress
book Pickwick the photographers
car
phone Sarah -- where did they find their caterers ?
how many people can we pack into the Tudor Room at the Dunkery Hotel ?
order a cake
find suitable Hotel for staying guests
```

The opposite symbol is <, which stands for 'less than'.

```
search action$<>"p" ↵
```

3 Type `search action$<>"p"` and press **ENTER**
The first record with anything other than `p` in the `action$` field appears:

```
see Vicar
```

```
seemore ↵
```

4 Use `seemore` to display the remaining records, which should consist of:

```
don't forget buttonholes
white shoes
buy a dress
has Nigel got a morning suit ?
make a present list
don't forget needle and thread
```

The symbols <> together stand for 'less than or greater than', in other words, 'not equal to'.

Relative operators

Relative (or relational) operators can be used to compare or check the relationship of one value (numeric or string) against another.

For example:
= tests if the two values are *equal*
> tests if the value on the left is *greater than* that on the right

The full range of relational operators is:

=	equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
<>	not equal to

Note that ><, => and =< will not work as combinations.

If you have trouble telling < and > apart, remember that the value on the *small* (pointed) side of the symbol must be *smaller* than the value on the larger (open) side if the expression is to evaluate as true.

Improving SEEMORE

The procedure `seemore` is still rather clumsy – the `if` clause tests for a condition (ie. found or not) which could be tested for by the `while` command. In other words, the loop should continue while `found()` is true, and returns the value 1.

`edit` ↵ TAB TAB

1 Use `edit` and TAB to highlight `seemore`

↓ ↓ ↓ F3 c

2 Cursor down to the `if` line and press F3 then `c`

↓ ↓

3 Cursor down twice more to mark the other two lines of the `if` clause for deletion. If you go too far, cursor back in the opposite direction.

↵

4 Press ENTER to cut all three lines, `if,stop,endif`

↑ F5 SHIFT ↓ CTRL ↑ `found()` ↵ ESC

5 Use F5 to change the line `while 1=1` to read `while found()`

This is the equivalent of `while found()=1`, since both the function on its own and the expression will return the value of 1 as long as `found()` is true.

The procedure now looks like this:

```
proc seemore
  while found()
    continue
    sprint
    endwhile
  endproc
```

`save` ↵ `seemore` ↵ F5 SHIFT ↓ ↓ `mdv1_` ↵

6 Save the procedure to the program `seemore`, and to `mdv1_seemore`.

This procedure only starts if `found()` is true and a match has been made. It then carries on looping only while `found()` is true after each `continue`. It will therefore stop after the last record which matches.

Finding part of a field with the string slice

Miranda wants to find out what she has to do in June.

`search date$="06"` ↵

1 Type `search date$="06"` and press ENTER
The first record, with `04` at the start of the date, appears.

The `search` command has failed despite the fact that there are June dates in the file. This is because the value being searched for must match the entire field *exactly*. The string `06` does not equal `06/01`.

It is possible to match on only part of a text string by telling ARCHIVE the positions of the characters you are interested in. The `date$` field has the month as the first and second characters, and the day as the fourth and fifth. The third position is occupied by the divider, a slash (`/`).

The slash is there simply for cosmetic reasons, as `06/01` is easier to read as a date than `0601`. The slash has no other function, but must be taken into consideration.

```
search date$(1 to 2)="06"↵
```

2 Type `search date$(1 to 2)="06"` and press **ENTER**

```
seemore↵
```

3 Use `seemore` to check the results.
Only the two records where the `date$` characters 1 and 2 are `06` have been matched.

The string slice

The string slice is effectively a function (although not classified as one) which returns a value representing only part of a larger string.

Numbers in brackets following the string (or string variable) designate which characters form the substring. The first character in the string is character 1. The positions for starting and ending the slice can be designated in a variety of ways:

For example, if the text field `note$` were to contain the value `see Vicar`:

```
print note$(5)          will print V
print note$( to 3)      will print see
print note$(2 to 5)     will print ee V
print note$(5 to )      will print Vicar
```

Finding ranges of records

Miranda now asks ARCHIVE to show her the memos for the first week in April.

To find records whose dates cover a period more or less than a month, ARCHIVE must be given a range of dates.

A range of values includes every value from a given starting value up to a given final value.

Ranges are specified by giving the first and last dates to be included, using the > and < signs to catch everything in between.

```
search date$>="04/02" and date$<="04/08" ↵
```

1 Type `search date$>="04/02" and date$<="04/08"` and press **ENTER**

These dates represent the first week in April. The first record, set for Sunday the 8th of April, appears:

```
see Vicar                                                    04/08
```

```
seemore ↵
```

2 Type `seemore` and press **ENTER** to see the rest of the records selected. All the memos dated the 5th of April appear:

```
ask Felicity to be a bridesmaid                              04/05
ask Jayne to make their dresses                              04/05
ask Mrs. Glazer to arrange the flowers                       04/05
has Nigel got a morning suit ?                               04/05
book Pickwick the photographers                              04/05
car                                                          04/05
phone Sarah - where did they find their caterers ?           04/05
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?                                                      04/05
order a cake                                                 04/05
find suitable Hotel for staying guests                       04/05
make a present list                                          04/05
```

The use of `and` in the expression forces the matching date to be *both* greater than or equal to the 2nd, *and* (ie. as well as) less than or equal to the 8th. The 14th, for example, is greater than the the 2nd, but not less than the 8th, and so does not match.

Searching strings with the INSTR() function

Nigel wants to check his involvement in the pre-wedding preparations. He has to find any notes in the `memos` file which contain his name.

The `search` command is unlike the `find` command in another significant way. It compares and matches text exactly with an entire field (or a specific part of it, if you use a slice). The command `find` matches text anywhere in any field.

To find text anywhere in a specific field, use the function `instr()`. The word is an abbreviation of `in string`.

This function works with `search` exactly like a `find`, but only on one field at a time.

```
search instr(note$,"Nigel") ↵
```

1 Type `search instr(note$,"Nigel")` and press **ENTER**
The first memo with Nigel's name in it appears:

```
has Nigel got a morning suit ?
```

```
seemore ↵
```

2 Use `seemore` to check for other matches. There are none.

The INSTR() function

The function `instr()`

- must have two strings inside the brackets, separated by a comma. They may be any string expression : variables, such as a field name; literals, between quotes; functions which return strings; or combinations of them.
- returns a value of 0 if the second string does not exist somewhere in the first. If the second does appear somewhere in the first, `instr()` returns the position of the first matching character in the longer string.

  For example, `print instr("see Vicar","Vic")` displays the number 5.
- evaluates as true (greater than 0) when the substring is found.

Note that the `instr()` function *does* distinguish between upper and lower case, like `search` but unlike `find`. It is said to be 'case specific'.

The SELECT command

Miranda wants to see all those records where she has to do something before May.

So far, the `search` command has been used to find records which match a certain requirement simply so that they can be looked at. To use them for anything else (such as listing or updating) would involve including commands inside the loop which check `found()`. Every matching record would thus be processed.

It is often easier to select the required matching records first, and process them as you wish afterwards.

```
select date$<"05/01" ↵
```

1 Type `select date$<"05/01"` and press **ENTER**
The following record appears:

```
see Vicar
```

`first` ↵ 2 Type `first` and press **ENTER**
The record does not change. The `select` command has left the file positioned on the first record which matches, rather than the last, as `search` might after finishing.

`zoom` ↵ 3 Type `zoom` and press **ENTER**
Notice that the record numbers printed at the bottom of the screen start at 0 and ascend in sequence up to 13.

`print count()` ↵ 4 Type `print count()` and press **ENTER**
ARCHIVE clears the screen and shows the figure 14, despite the fact that there should be 16 records altogether in the file.

The `select` command excludes all records which do not match, and ARCHIVE simply ignores them. It is as if they had never existed.

`screen` ↵ 5 Type `screen` and press **ENTER**

`select action$="p"` ↵
6 Type `select action$="p"` and press **ENTER**

`zoom` ↵ 7 Type `zoom` and press **ENTER**
This time only records that were already in the previous selection and have `p` in the `action$` field are selected. There are 9 records numbered from 0 to 8.

The resulting selection could have been achieved with a single command:

```
select date$<"05/01" and action$="p"
```

`first` ↵ 8 Use `first` to go back to the first record in the selection.

`listmemos` ↵ 9 Type `listmemos` and press **ENTER**
The selected records are listed line by line in the DISPLAY AREA:

```
note$                                                    date$    action$
ask Felicity to be a bridesmaid                          04/05    p
ask Jayne to make their dresses                          04/05    p
ask Mrs. Glazer to arrange the flowers                   04/05    p
book Pickwick the photographers                          04/05    p
car                                                      04/05    p
phone Sarah -- where did they find their caterers ?      04/05    p
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?                                                  04/05    p
order a cake                                             04/05    p
find suitable Hotel for staying guests                   04/05    p
```

Once a file has been selected, the selected records remain the only currently accessible records until the file is closed or reset.

A selection which matches no records at all makes all records inaccessible, as if there were no records in the file.

Selections affect most commands, even `all`, but do not affect `close`. All the original records are saved onto tape (if opened with `open`) when a selected file is closed.

`select` marks every record which matches as 'logically present'. Although every record is still physically present in the file after `select`, ARCHIVE behaves as if it 'thinks' only the selected records are there.

Resetting the file

`reset` ↵

1 Type `reset` and press **ENTER**

`listmemos` ↵

2 Type `listmemos` and press **ENTER**
This time all 16 records of the original file are listed on the screen. The order may not be the order they were originally entered, (before they were first selected) but they are all there.

The command `reset` has the effect of cancelling any selections made.

Unlike `select`, the `reset` command does not make the first record the current record.

Remember to use `reset` even if you mistakenly give a `select` command which matches nothing, as the file will then contain no records: any further, corrected selections will obviously fail, as will all record-related commands.

Note: the standard display will display the last record in the file in the event of a failed `select`. This is misleading, as in fact there are no selected records in the file. The `count()` function returns the value 0.

Closing a file which was opened with `open` has the effect of resetting it. All records, not just the selected ones, are saved back onto the tape.

Note: `reset` also cancels the effect of the command `order`, introduced in the next chapter.

FURTHER SELECTIONS

Selecting with AND

Miranda wants to see the names of guests coming to the wedding from London. One of her old school friends who lives there doesn't have a car, and might be able to get a lift from someone who does.

`look↵guests" logical "g↵`

1 Open the `guests` file with the command `look`.

```
display ↵
```

2 Display the first record.

The guests file cannot reveal who has or hasn't got a car, because no such information is stored there; instead, Miranda selects everyone in London with less than 4 in their party. She can then assess from memory whether they are likely to be driving.

The layout of different addresses means that the word `London` doen't always have to be in the same field. It is likely to appear on one of the address lines after the first one, so she opts for checking lines 2 and 3.

Logical operators

The logical operators are `not`, `and` and `or`.

Logical operators are distinct from relative (`= < >`) and arithmetic (`+ - * / ^`) operators, which have already been described.

`not` is called a *unary* operator, since it can be used on its own, like the minus sign. For example, `not found()` and `-6` are both valid expressions, but other operators (`= + < *` and so on) need to have expressions on both sides of them.

`not` can thus be used in the middle of expressions as well as at the start.

Note: early versions of ARCHIVE do not always evaluate `not` expressions correctly, and should be avoided eg. `if true and false<>0` instead of `if true and not false`.

The word logical refers to a form of algebra where the result of every equation is either simply true or false. This is called Boolean algebra, after its inventor. Boolean logic, using operators like `not`, `and`, and `or`, is used to solve the equations.

```
select partyof<4 and address2$="London" and address3$="London"
```

3 Type `select partyof<4 and address2$="London" and address3$="London"`

Note that the `partyof` field is numeric, so the value to be compared with it is not in quotes.

↵ 4 Press **ENTER**

The last record in the file is displayed, and it is not a London one.

The selection failed because the word London only appears once in an address. It cannot be both on line 2 and on line 3.

Both sides of an and must be true for the whole and expression to be true.

The word and is called a logical operator. As usual, this has nothing to do with logic in any normal sense of the word.

Selecting with OR

```
reset ↵
```

1 Type reset and press **ENTER**

```
select partyof<4 and address2$="London" or address3$="London" ↵
```

2 Type select partyof<4 and address2$="London" or address3$="London" and press **ENTER** or use **F5** and edit the line.

Either half of an or expression can be true for the whole or expression to be true. That is, if either address2$ equals London OR address3$ equals London, the expression is true.

In this case, if neither line equals London, the record is not selected. If at least one does, the record will be selected. A London address appears in the display:

```
John and Sue Heath  43b Clarkson Road London  SW16   y  2
```

```
zoom ↵
```

3 Use zoom to see the rest:

```
Sally Jones and Susan Clark 3b Academy Mews   Chelsea London SW3 y 2
```

Remember that if you accidentally entered any literal spaces at the end of an address field (eg. "London ") they will not be visible but this selection will not match and include them. You can check for final spaces by using the string slice and len() function, eg:

```
search address1$(LEN(address1$))=" "
```

The function len() returns a number representing the length of the string between its brackets, eg:

```
print len("Hello")
```
displays 5

CHECKING DATA WITH LOGICAL EXPRESSIONS

Many kinds of data (such as dates and codes) need to be entered accurately to be of any use.

Logical expresions can be used to check that the data is of the correct type, length, and in the right range of possible values.

If it is wrong, the input is fetched again.

A range-checking procedure

```
proc inputletter
  let ok=0
  while not ok
    input "Initial ?";init$
    if init$>="A" and init$<="Z" and LEN(init$)=1
      let ok=1
      else
      print "Initial must be a capital letter"
      endif
    endwhile
  endproc
```

This procedure loops until the correct input (a single letter) has been entered. The variable `ok`, initially set to false (0), is used as a flag to indicate correct or incorrect data. When `ok` is set to true, the loop finishes.

A more compact version of the same procedure uses the value of the expression itself (true or false, 1 or 0) to set `ok`:

```
proc inputletter
  let ok=0
  while not ok
    input "Initial ?";init$
    let ok=(init$>="A" and init$<="Z" and LEN(init$)=1)
    endwhile
  endproc
```

The most compact version of this procedure dispenses with the variable `ok` altogether – the input variable itself is preset to a wrong value and the loop continues while it remains wrong:

```
proc inputletter
  let init$=""
  while init$<"A" or init$>"Z" or LEN(init$)<>1
    input "Initial ?";init$
    endwhile
  endproc
```

Two possibles

If input data must be only one of two possible values, remember to use `or`:

```
if init$="Y" or init$="N"
  let ok=1
  endif
```

This passes `init$` as ok if it is `Y` or `N`. Using `and` here causes everything to be rejected, as a character cannot have two values at once.

Note that if you reverse the question (and use <> instead of =), then you must also *reverse* the logical operator:

```
if init<>"Y" and init$<>"N"
  let ok=1
  endif
```

This passes everything except `Y` and `N` as ok.

Several different possibles

If there are several possibilities and they do not fall neatly into sequence as a range of values, use the function `instr()`:

```
proc authorise
  let ok=0
  while not ok
    input "Director's initials: ";init$
    let init$=upper(init$)
    if instr("JB/HS/AM/MD/SS",init$) and len(init$)=2
      let ok=1
      endif
    endwhile
  endproc
```

Note that the slash mark has been used to prevent unwanted combinations such as BH, SA, MM etc. (the last letter of one correct option and the first of another) from being passed as `ok`.

To be really pedantic, you should exclude slashes from the input by adding another check to the start of the expression:

```
if instr(init$,"/")=0 and ........
```

Other methods of making data checks when there are many possible correct entries are discussed in Chapter 11.

Checking dates

A single complex expression can be used to check a date:

```
proc checkdate
  let getdate=1
  while getdate
    input "Date (as YY/MM/DD) ";date$
    let y=date$(to 2)
    let m=date$(4 to 5)
    let d=date$(7 to)
    let getdate=0
    if (d>29 and m=2) or (d>28 and m=2 and y/4<> INT(y/4)) or
(d>30 and (m=4 or m=6 or m=9 or m=11) or m<1 or m>12 or d<1 or
d>31
      let getdate=1
      endif
    endwhile
  endproc
```

This expression can be paraphrased as:

IF day>29 and month is Feb
OR day>28 and month is Feb and year is not leap (exactly divisible by 4)
OR day>30 and month is Apr, Jun, Sep or Nov
OR month<1 or month>12 or day<1 or day>31
THEN it's wrong, get the date again.

Note that this procedure also uses the flag (`getdate` in this case) in a different way. This time the loop ends when the flag is false. It is set to false inside the loop on the assumption that the input will be correct, and is only changed if it is not.

Reversing the logical outcome

You can completely reverse a complex expression without altering any operators in several ways.

- by using flags in different ways (as above)
- by bracketing the whole expression with `not` at the front
- by inserting an `else` on its own at the start of the `if` clause.

For example: To reverse

```
if keys$="X"
 stop
 endif
```

just add an `else` statement as follows:

```
if keys$="X"
  else
  stop
  endif
```

This last clause stops only if `keys$` is *not* `X`, and is the equivalent of:

```
if not(keys$="X")
  stop
  endif
```

Note: `not` does not always work correctly in early versions of ARCHIVE when used like this.

Notes on evaluating logical operators

- When evaluating expressions as true or false, ARCHIVE takes each operator in turn, and evaluates it before moving onto the next.
- Every operator has a priority in this process. ARCHIVE does not simply start with the first one on the line.
- Arithmetic operators (`* / + -`) come before relational operators (`= > <`), which in turn come before the logical operators (`not and or`).
- The operator `not` is evaluated before `and`, and `and` before `or`.

  Note: Unfortunately, early versions of ARCHIVE have a bug which causes it to ignore the priority of `and` over `or`.

  Avoid this problem by putting `and` before `or` in the line, or, better still, putting each pair of values inside brackets (this also makes complex expressions easier to read).
- Expressions in brackets are evaluated before expressions which are not. When there are brackets inside brackets, the expression in the deepest set is evaluated first.
- Brackets have the effect of overriding the normal priorities of operators.

Fully worked examples

When dealing with complex selections involving relational and logical operators, it can help to write the expression down and evaluate it abstractly, one operation at a time. Test the expression against sample information, some of which should fail the test (if the test has been written correctly), and some of which should pass.

In the following examples note that each operator has a pair of expressions connected with it. Each pair is evaluated in turn, according to the priority of its operator, The result is always either true (greater than 0) or false (0).

```
Example data:
address2$     :London
address3$     :
partyof       :3
```

In all the following examples, the relative operators (<, >, =) are evaluated first, but are glossed over here to emphasize the sequence of logical operators.

Example 1: OR

Evaluates or

```
address2$="London" or "address3$="London
true      or      false
      =true
```

ARCHIVE asks the question: is at least one of the equations true? If it is, the whole expression is true.

Either side, or both sides, of an or expression may be true to evaluate as true.

Example 2: AND

Evaluates and

```
address2$="London" and "address3$="London"
true      and      false
     =false
```

ARCHIVE asks the question: are both the equations true? If they are, the whole expression is true. Remember that both sides of an and expression must be true to evaluate as true.

Example 3: OR, AND

Evaluates first and then or

```
address2$="London" or "address3$="London" and
partyof<4
     false      and      true
     true      or      false
     =true
```

ARCHIVE first asks the question: are both the comparisons on either side of `and` true? If they are, the whole `and` expression is true.

ARCHIVE then asks if either the whole `and` expression or the equation on its other side is true. If one or both are, the whole expression is true.

Note: this order of `and` and `or` does not actually work as it should in early verions of ARCHIVE. Put the `and` expression in brackets to overcome this.

Example 4: (), OR, AND

Evaluates brackets `()`, then `and`, then `or`

```
(address2$="London" or "address3$="London") and partyof<4

    (true     or     false)

    true     and     true

    =true
```

ARCHIVE first asks the question: is either of the equations in the brackets true? If at least one is, the whole `or` expression is true.

ARCHIVE then asks if both the `or` expression and the comparison on the other side of `and` are true. If they are, the whole expression is true.

Remember that brackets override the usual priorities. Putting an expression inside brackets causes it to be evaluated before expressions outside them.

Avoiding logical operators

If you find logical operators confusing, there are usually ways to avoid the problem.

One technique is to use `select` repeatedly, further selecting the last selection each time. Do not use `reset` between successive selections.

Another way is (for example) to temporarily run all the address lines together as a single string for checking with `instr()`. This also saves typing the key word, `London`, several times, eg:

```
select instr(address2$+address3$,"London") and partyof<4
```

This matches any instance of London in the second and third lines of the address where the party is also less than 4. It also has the advantage of finding the word if it is not the first word on the line.

Remember that text may be added together to make longer text. The function here evaluates the two address lines as if they were a single string, eg. `instr("ClaphamLondon","London")` is true.

There is another, much faster way of finding and grouping records, especially in files with many records. This is done by using the commands `order` and `locate`, which are the subject of the next chapter.

Tidying up the tape

`dir` ↵ ↵ 1 Do a directory of drive 2 to check the date on the `_lab` file.

`dir` ↵ `mdv1_` ↵ 2 Do a directory of your backup tape, in drive 1, to check the date on the `_lab` file to make sure it is the correct backup tape. If it is the same as your working tape, use your other backup tape.

`new` ↵ 3 Clear all procedures and data files from temporary memory by typing `new` and pressing **ENTER**

`backup ...` 4 Backup files created or changed since the tape was last used.

`save "Apr6_lab"` ↵ F5 SHIFT ↓ ↓ `mdv1_` ↵
5 Save label files with today's date onto both the working tape and the backup tape in Microdrive 1.

6 `kill` the old `_lab` files on both tapes.

SUMMARY

Miranda uses the `search` command to find notes in the `memos` file which say she needs to make a phone call (where the `action$` field is `p`). To see which records match without repeating the command `continue`, she writes a procedure called `seemore`. This procedure at first uses a *forever* loop, which is a loop (eg. `while/endwhile`, `all/endall`) which is always true. In this case ARCHIVE is told to go on using the `continue` command while `1=1`, ie. forever.

Miranda improves the `seemore` procedure by using the `found()` function. This function has two values, 1 or 0. It will be 1 if the record is found, and 0 if not. In `seemore` it is used with the `while, endwhile` loop. `seemore` goes on using the command `continue` as long as `found()` is equal to 1. As soon as it becomes 0 (when there are no more records matching the search), the procedure ends.

After this, Miranda uses the 'string slice' to find memos dated June. The string slice finds text or numbers at a specified position within a field. The `instr()` function finds text or numbers at any position within a field. She then goes on to use the `select` command to limit the available records to those which match

certain criteria. When the `select` command is used, only the selected records are then available for looking at or altering. To regain access to the other records in the file, the `reset` command must be used.

Using the `select` command and the `guests` file, Miranda now finds all those guests who live in London and have fewer than four in their party. She uses the *logical operators* `and` and `or` to do this.

Helpful hints

- If you don't appear to have the correct number of records in your file, you may have forgotten to `reset` the file after using the `select` command.
- Put each pair of values in brackets when using expressions. This is easier to read and ensures correct evaluation.
- Use validations to check data entry. If data is checked as it is entered, mistakes which might cause untold problems at a later date can be avoided.
- The function `instr()` can be used to replace the logical operator or when validating input.

Sorting Information

One of the most tedious clerical tasks ever inflicted on human kind is the sorting of records into a specific sequence, such as alphabetical order.

Sorting is often vital for fast retrieval or analysis of groups of similar records.

The more records there are, the worse the burden – and the greater the probability of mistakes. In a large system, a mis-filed record can be lost forever. All new records have to be put into order before they can be efficiently filed, and, once done, no one wants to change the order.

If the records occasionally need to be seen in a different order, complete duplicate cross-referencing systems are usually set up, as continually re-sorting the paper records is clearly impractical, if not impossible.

ARCHIVE orders records effortlessly and efficiently, into as many sequences as you want and as often as required.

This chapter shows you how to put your data into a specific order, and how to find it again – easily.

The following features of ARCHIVE are introduced:

- The ARCHIVE commands:

  `order` – The use of `order` to sort records into a defined order

  `locate` – The use of `locate` to find a record in an `ordered` file

Points to remember

- Use 'composite keys' to avoid the space penalties of using too many levels of order. Do this by storing them as a separate field beforehand.

- An indexed find, using `locate` with an `ordered` file, is much faster than using `find` or `search`.
- If the number or string to be matched is not found, the next record in sequence becomes the current record.

Example: Putting the guests in order

Starting up

If ARCHIVE is not loaded, load it now and put your data cartridge into Microdrive 2.

```
load ↵ seemore ↵
```

1 Load the program file `seemore`, created in the last chapter.

```
look "memos" logical "m" ↵
```

2 Open the `memos` file with the command `look`.

```
look "guests" logical "g" ↵
```

3 Open the `"guests"` file with the command `look`.

SORTING AND SEQUENCING

One of the big headaches for Miranda and Joyce is the seating plan for the reception. How should they arrange their guests so each one is sitting next to someone they will find interesting to talk to?

To this end, they endeavour to ensure that people who arrive together are not seated together.

They produce a list, ordered on the number of guests, to simplify the breaking up of large groups.

THE ORDER COMMAND

Information often needs to be sorted into a specific sequence, regardless of its original order.

Telephone directories are usually sorted into surname order. Stacks of mail awaiting delivery at the sorting office will be sorted by house, street and area. Diaries and calendars are in date sequence. The bank manager's list of overdrawn customers might be in account number sequence, the lowest number coming first. Score lists are usually ranked by score, the highest score coming first.

This is especially useful when reading from printed lists since records can be looked up quickly. If they don't appear where they ought to be in the sequence, then they are not in the list (or were mistyped and so misplaced).

- ***The same information often needs to be seen in more than one order.***

Accounting information is usually listed once by customer and once by date. This makes it easy to check a telephone query from a customer, as well as facilitating an auditor's task.

Other typical orders allow, for example, finding a book by author or by title, or finding a telephone number by surname or by occupation (as in the *Yellow Pages*).

Ordering records also has the effect of grouping similar records together, making it easy to answer questions like: How many books do we have on this subject, or by this author? How much business was done with this customer on that day?

Although the `find`, `search` and `select` commands offer a variety of ways for finding and selecting records, none of them puts them into any recognisable or useful sequence.

This is especially significant in ARCHIVE, since records are stored in the most efficient way in terms of space, and will not always be in the sequence in which they were inserted. Their relative position can change after commands like `select` and `reset`.

The command `order` puts records in order using one or more fields.

`reset` ↵ `display` ↵

1 Reset the guests file and re-display it if necessary.

`order` ↵ `partyof;a` ↵

2 Type `order partyof;a`. Note the semicolon (`;`) after the field name, and the letter '`a`' following it. This sorts the file by the contents of the `partyof` field, in ascending order. This means the lowest figure comes first, and the highest last.

↵ 3 Press **ENTER**. The display changes to show a record which has only one person coming from that address.

`zoom` ↵ 4 Use `zoom` to see the rest of the file, watching the `partyof` field and also the record number. The records should be displayed in the following order:

```
name$                          partyof
Mrs Agnes Blake                 1
Dame Edith Throgmorton-Dunne    1
Mabel Pew                       1
Bill Hackney, Drama Critic      1
Jon Ducane                      1
John and Sue Heath              2
Philip and Brenda Moore         2
Sally Jones and Susan Clark     2
Lt-Col and Mrs Anthony Laurence 3
Joshua Grimshaw                 3
The Paradise Posse              3
Arthur and Janice Blake         7
```

When ARCHIVE orders on a field, that field in successive records is always in either ascending or descending order. However, within the groups of records with the same information in the ordered field, (eg. all records where `partyof = 3`) the records could be in a different order to the order in which they were originally entered into the file.

Note: Early versions of ARCHIVE do not order negative numbers correctly. In an ascending order, minus values are placed at the end, which is obviously wrong.

Like the `select` command, `order` numbers all the records from zero onwards according to the new sequence, regardless of the original record number.

- The command `reset` resets an `ordered` file as well as a `selected` one.

Automatic ordering

Once a file has been `ordered`, any record added by `insert` or `append` is automatically placed in its correct position in the sequence. This is also true when a KEY field (a field used with the `order` command) is `updated` or `altered`.



Harold gets the bill

THE LOCATE COMMAND

The command `locate` finds records in an ordered file.

Ordering numeric fields

```
locate 3 ↵
```

1 Type `locate 3` and press **ENTER**.
Note that the 3 is not in quotes, as `partyof` is a numeric field.

The first record in the file with a party of 3 appears in the display:

```
Lt-Col and Mrs Anthony Laurence     Overton Grange  Wideacres Lane
Upper Brackett    Cumbria           3
```

Points to remember

- `locate` can only be used to match the contents of the field which was used to `order` the file, so there is no need to specify the field again.
- Like `find` and `search`, the command `locate` finds the first instance of a certain value. Like `search`, the match is specific to a certain field, and if the value is a string, upper and lower case are treated as different.
- Unlike `search`, `locate` matches only as many letters as given with the command, eg. `locate "N"` matches `Nigel`.

  This is only true of strings. Numbers must match in their entirety.
- If `locate` does not match a number or the beginning of a string, the next record in sequence becomes the current record. In ascending order, this is the next highest. In descending order, it is the next lowest.
- The function `found()` works with `locate` in ARCHIVE version 2 exactly as it does with `find` and `search`. That is, if an exact match is found, it returns the value 1.
- The `continue` command does not work with `locate`. Any more records with the same value can be found by using a `while-next` loop. They will be immediately after the first one found, because `order` has grouped all records with similar values together by putting them in sequence.

To see the remaining records with a 3 in the `partyof` field:

```
while partyof=3:sprint:next:endwhile ↵
```

1 Type the above and press **ENTER**.
The following records are displayed until the first `partyof` which is greater than 3 is found. The first record which has a `partyof` field greater than 3 causes the loop to fail, and is automatically displayed afterwards by the standard display.

The `locate` command is also unlike `find` and `search` in that if it does not find an exact match, it finds the first record which follows where the matching record would have been.

```
order ↵ partyof;d ↵
```

2 Type `order partyof;d` and press **ENTER**.
In this case, the `d` stands for descending, as opposed to a for ascending.

```
zoom ↵
```

3 Use `zoom` to see the file.
Once again the records have been resequenced and renumbered, this time in descending order, with the highest value first. The records should be in this sequence:

```
name$                              partyof
Arthur and Janice Blake             7
Lt-Col and Mrs Anthony Laurence     3
The Paradise Posse                  3
Joshua Grimshaw                     3
Sally Jones and Susan Clark         2
Philip and Brenda Moore             2
John and Sue Heath                  2
Mrs Agnes Blake                     1
Jon Ducane                          1
Dame Edith Throgmorton-Dunne        1
Bill Hackney, Drama Critic          1
Mabel Pew                           1
```

In the guests file there are groups of 1, 2, 3 and 7 coming to the wedding. Try the following commands:

`locate 8` finds the first record, with 7 in the `partyof` field

`locate 5` finds the first record lower than 5, which is 3

`locate 0` finds the last record, since there is none lower

Note: A `locate` command which does not match exactly finds the next value higher than itself if the file is sorted in ascending order.

The `locate` command works much faster with large files than either `find` or `search`. Both of them start at the first record and examine every record one after the other to see if it matches, because a matching record could be anywhere in the file.

`locate` jumps immediately to the place where the match should be found. It can do this because the file has been put into sequence, and the matching records will not be anywhere else in the file. ARCHIVE knows where the record should be.

Ordering text fields

`order name$;a` ↵

1 Type `order name$;a` and press ENTER.

`zoom` ↵

2 Use `zoom` to see the sequence.
The records are sorted into alphabetical order, but not strictly by name, since the first word in `name$` can be a title like `Mr` or `Mrs`:

```
name$                           address1$
Arthur and Janice Blake         131 Jubilee Terrace
Bill Hackney, Drama Critic      The Marston Gazette
Dame Edith Throgmorton-Dunne    The Boundaries
John and Sue Heath              43b Clarkson Road
Jon Ducane                      Northern Lights Dance Company
Joshua Grimshaw                 The Old Rectory
Lt-Col and Mrs Anthony Laurence Overton Grange
Mabel Pew                       12 Acacia Avenue
Mrs Agnes Blake                 Cherry Cottage
Philip and Brenda Moore         12 Queens Gardens
Sally Jones and Susan Clark     3b Academy Mews
The Paradise Posse              The Athenaeum Theatre
```

It is more useful to sort on a fixed item, such as the surname. This can only be done if the surname appears at the start of the field. Rather than store the first and surnames back to front, this can be achieved where needed by having one field for first names and another for surnames. Join the fields together for printing.

Note: ARCHIVE only sorts on the first 8 characters of a text string – *any character(s) after the eighth will have no effect on the sequence*. For example, two records sorted on company name fields beginning with the word 'British' could appear sorted in any order, with 'British Zinc Ltd.' coming before 'British Aardvarks Ltd'.

Ordering by date

`look "memos" logical "m"` ↵

1 Open the `memos` file if it is not already open.

`use "m"` ↵ `display` ↵

2 Use and display the `memos` file.

`order date$;a` ↵

3 Type `order date$;a` and press ENTER.
The memos are now in date sequence, as long as the dates have

been entered in the American way, with the month first, as explained in chapter 5:

```
note$                                                    date$
book Pickwick the photographers                          04/05
phone Sarah - where did they find their caterers ?       04/05
car                                                      04/05
order a cake                                             04/05
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?                                                  04/05
find suitable Hotel for staying guests                   04/05
make a present list                                      04/05
has Nigel got a morning suit ?                           04/05
ask Felicity to be a bridesmaid                          04/05
ask Jayne to make their dresses                          04/05
ask Mrs. Glazer to arrange the flowers                   04/05
see Vicar                                                04/08
buy a dress                                              04/14
white shoes                                              04/14
don't forget needle and thread                           06/23
don't forget buttonholes                                 06/23
```

Note: String slices cannot be used with the `order` command, so attempting to order on British style (day, month, year) dates cannot be done. For example, the command:

```
order date$(4 to 5)+date$(1 to 2)
```

does not work. This is why the date is stored with the month first. If you need to sort by year, the year must precede the month.

`locate "06"` ↵

1 Type `locate "06"` and press **ENTER** to find the first memo dated June:

```
don't forget needle and thread
```

`locate "04/06"` ↵

2 Type `locate "04/06"` to find the first record after April the 5th.

```
see Vicar
```

Remember: up to the first 8 characters of a text field must be entered exactly as you wish the file to be sorted if you plan to use it with the `order` command.

This prevents ARCHIVE from ordering the file on dates created by the function `date()`. `date()` produces date strings (including the century) which are longer than 8 characters, eg: 1984/04/05. Use the string slice to remove the first two characters before updating the field, eg: `let date$=date()(3 to ):update`.

Order the file by the action code:

`order action$;a` ↵

3 Type `order action$;a` and press **ENTER**. The records are now sequenced according to the contents of the `action$` field.

Empty fields come first, followed by 'p' for phone and finally 's' for shop:

```
note$                                                    action$
don't forget buttonholes
don't forget needle and thread
see Vicar
has Nigel got a morning suit ?
make a present list                                          p
ask Felicity to be a bridesmaid                              p
ask Mrs. Glazer to arrange the flowers                       p
ask Jayne to make their dresses                              p
car                                                          p
book Pickwick the photographers                              p
phone Sarah - where did they find their caterers ?           p
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?                                                      p
order a cake                                                 p
find suitable Hotel for staying guests                       p
white shoes                                                  s
buy a dress                                                  s
```

`locate "p"` ↵

4 Type `locate "p"` and press **ENTER**.
The first `p action$` field appears. Use `zoom` to see all the phone calls to be made.

`locate "S"` ↵

5 Type `locate "S"` (Note that the 'S' is a capital letter.) and press **ENTER**.
A record with `p` appears in the display. Text fields are sorted alphabetically according to the ASCII character sequence (*see Appendix*). In the ASCII codes, all capital letters come before any lower case letters. `locate` has here found the first lower case record, since there are no upper case records at all.

Note that using `locate` with a lower case letter in a file ordered on names (which usually start with a capital letter) will have the opposite effect: the last record in the file will be found, eg. `locate "andy"` might find "Zachariah".

Ordering on more than one field

Miranda would prefer a more useful sequence that sorts the memos by date and then by action code within each day.

`order date$;a,action$;a` ↵

1 Type `order date$;a,action$;a` and press **ENTER**.
Note the comma dividing the two fields, and that each has been

separately marked as in ascending sequence. It is possible to freely mix string and numeric fields, in ascending or descending sorts, up to eight levels, as long as they are divided by commas.

```
zoom ↵
```

2 Type `zoom` and press **ENTER** to see the rest of the records, which should be in the following order:

```
note$                                                    date$
action$ has Nigel got a morning suit ?                   04/05
make a present list                                      04/05
book Pickwick the photographers                          04/05 p
phone Sarah - where did they find their caterers ?       04/05 p
order a cake                                             04/05 p
find suitable Hotel for staying guests                   04/05 p
how many people can we pack into the Tudor Room at the Dunkery
Hotel ?                                                  04/05 p
car                                                      04/05 p
ask Mrs. Glazer to arrange the flowers                   04/05 p
ask Felicity to be a bridesmaid                          04/05 p
ask Jayne to make their dresses                          04/05 p
see Vicar                                                04/08
buy a dress                                              04/14 s
white shoes                                              04/14 s
don't forget needle and thread                           06/23
don't forget buttonholes                                 06/23
```

Memos for April the 5th with no action code come before those for the same day with a 'p'.

```
locate"04/05","p" ↵
```

3 Type `locate "04/05","p"` and press **ENTER** to locate the first memo requiring phoning on April the 5th.

Secondary sorts

The command `locate` can be used to pinpoint a secondary level of sort within a major one, as long as the values to be located correspond with the the sequence of the fields used in the `order` command. In 3, above, the date must precede the action code since this was the `order` sequence in 1.

You do not have to use more than one field with a `locate` command on a file which is ordered on more than one. Records can still be found based on the first field only.

Composite keys

It is not a good idea to `order` on multiple fields if you have lots of records or several files open. Each level of ordering uses up more

memory, and reduces the space available for other files. It also limits the maximum records a file can hold. (See the appendices for a fuller explanation of this.)

You can create a single *composite key* (of up to 8 characters) by duplicating parts of fields in a special key field (using concatenation and the string slice), eg:

```
let key$=company$( to 4)+name$( to 4):update:order
key$;a
```

This wastes some space, but does not interfere with the maximum number of records, nor the number of open files.

Saving the order

If you close an ordered file which was opened with the `open` command, the file will be saved onto tape in the ordered sequence. When it is next opened it will still be in the saved order, but can be set to its original sequence by the command `reset`.

The automatic ordering of inserted and changed records is still effective with an ordered file which has been saved and opened again.

This is not true when you close a file which was opened with `look`, as nothing is saved back onto tape.

How ORDER works

The `order` command uses an index to find records. This is stored in memory, separately from the records themselves, and acts exactly like an index in a book.

To find any mention of the command `input` in this book, you could scan every page (a sequential search), but this would take a long time. Alternatively, it only takes a moment if you use the index. It contains the word `input` sorted alphabetically, and lists the page numbers where you should look.

ARCHIVE does exactly the same with records. It stores the first eight characters of string fields in order, together with the place the record can be found, in an index. This is why `locate` is so much faster with large files.

The price you pay for this facility is that each index entry uses space in memory, and the space available is limited. The more files you open, and the more levels you order on, the more space you use.

Tidying up the tape

```
kill ↵ zoom_prg ↵
```

1 Delete the `zoom` program from the tape, as it has now been superseded by `seemore`.

`dir` ↵ ↵

2 Do a directory of drive 2 to check that you now have the following program files on tape: `easy_prg`, `wedfiles_prg`, `dummy_prg`, `lists_prg` and `seemore_prg`.

Check the date on the `_lab` file in drive 2.

`dir` ↵ `mdv1_` ↵

3 Do a directory of your backup tape, in drive 1, to check the date on the `_lab` file to make sure it is the correct backup tape. If it is the same as your working tape, use your other backup tape.

`new` ↵

4 Clear all procedures and data files from temporary memory by typing `new` and pressing **ENTER**.

5 Backup files created or changed since the tape was last used.

`save"Apr6_lab"` ↵ F5 SHIFT ◗ ◗ `mdv1_` ↵

6 Save the label files with todays date onto both the working tape and the backup tape in drive 1.

7 `kill` the old `_lab` files on both tapes.

SUMMARY

Miranda tries out the `order` command, using it to put the records into various sequences. First of all, she puts the `guests` file into the order of the number of guests coming (`partyof`). By putting an `;a` after the name of the field to be ordered on, the file is sorted in ascending order. A `d` would sort it in descending order.

Miranda then uses the `locate` command to find guests with a specific number in the `partyof` field. `locate` can only be used with a file that has been ordered, and only on the field on which it has been ordered.

The second field Miranda tries the `order` command on is `date$`. She can do this, because she has entered the dates in the American sequence with the month first. After this, she uses the `action$` code to sequence the records. All the records with a null string in the `action$` field come before those with a `p`, which come before those with an `s`.

`order` can be used with either numeric or text fields, and can be used with more than one field, so Miranda finally orders the file on both `date$` and `action$`, so she can see what groups of tasks she has to do on each day.

Helpful hint

- Store dates in the American style: year, month, day (without the century (19); dates will then sort correctly into chronological order rather than text order.

CHAPTER 9

Maintaining Records

Most database operations need the same few jobs to be done over and over again. These are unique to the nature of the file or task, but are usually very similar. A library, for example, sends reminders for overdue books, and businesses send statements for overdue payments.

All files need, at some stage, to be added to or changed, counted or sorted, printed or displayed. These three areas are often referred to in computer jargon as input, processing and output.

The real benefits of ARCHIVE are gained when procedures specific to a particular task are all available at the same time and with the minimum effort. Such a program should also be essentially adaptable to any files or task.

In this chapter, you will put together a program to handle the finding, browsing and updating functions normally needed when maintaining records.

The following ARCHIVE features are introduced:

- The ARCHIVE commands:
  - `error` – catches errors without stopping program
  - `return` – causes a procedure to stop and return control to the procedure that called it
  - `else` – used with `if` to offer an alternative
- The ARCHIVE functions:
  - `getkey()` – causes ARCHIVE to wait for a key press
  - `errnum()` – returns the number of the last error
  - `rept()` – repeats a string a number of times
  - `str()` – converts a numeric expression to a string

Example: RSVP

The replies to the wedding invitations start to arrive, and Miranda updates the `guests` file to provide an accurate picture of numbers.

Taking each reply in turn, she first has to `find` and then `alter` the record to update the `coming$` and `partyof` fields.

She already has a few procedures for performing some of the jobs required, but wants to link them together to simplify the task.

She is also planning to cajole the twins to clear some of the backlog for her. She needs the program to be informative for the twins and safe for the data.

Instead of typing in commands directly on the command line, single keystrokes will get ARCHIVE to perform various procedures.

Starting up

1 If ARCHIVE is not loaded, load it now.

```
open ↵ guests" logical "g ↵
```

2 Open the `guests` file with logical name `g`.

```
display ↵
```

3 Display the `guests` file on the screen.

PROCEDURES TO LOG EXPECTED GUESTS

Before creating any procedures, Miranda thinks about what she wants them to do. She starts by writing this on a piece of paper, in the broadest terms, and then slowly translating this into an approximation of the terms used and steps taken by ARCHIVE.

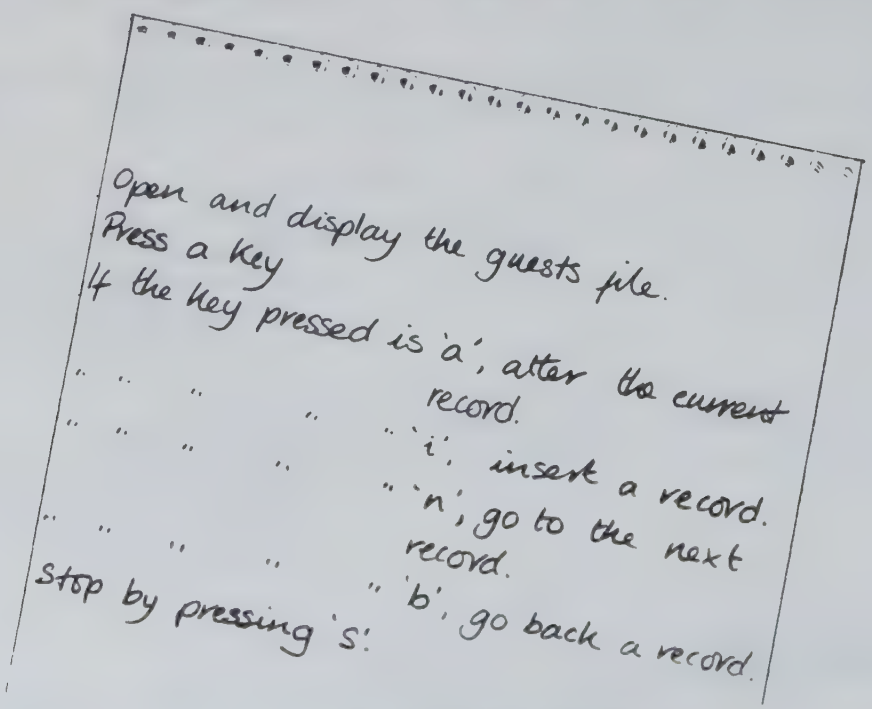


Figure 9.1 – Miranda's step by step analysis

She decides not to open and close the files every time the program is used, as she may be stopping to perform another task with them, and does not want to waste the time.

The outline she's written already looks very much like some of ARCHIVE's commands. She alters it to read:

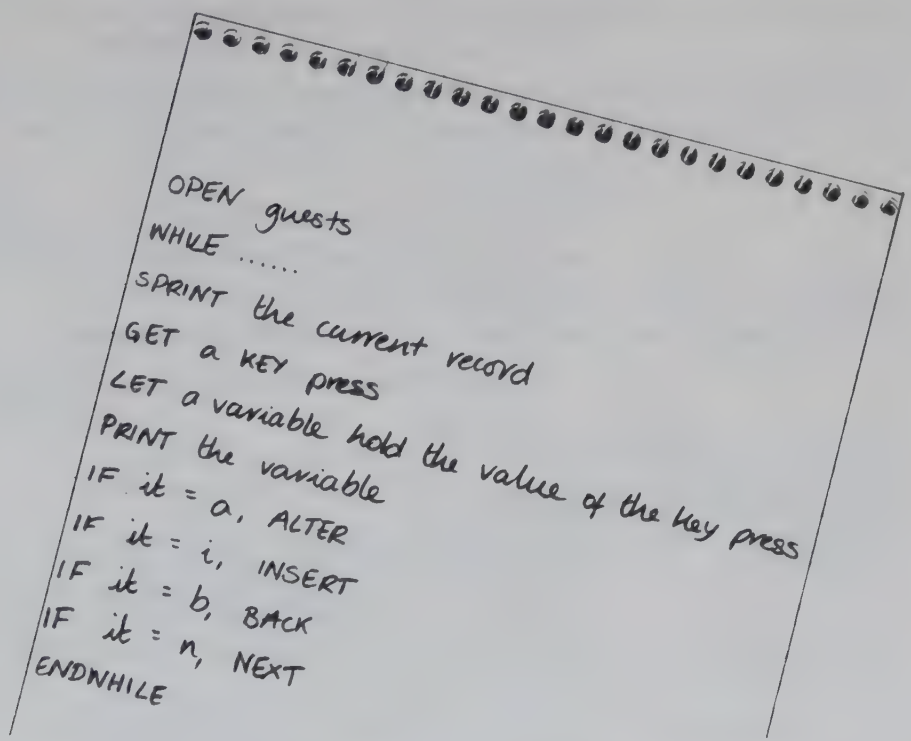


Figure 9.2 – A more formal analysis

She knows that she will soon need other options, and decides to break down the task into two procedures for clarity.

She starts work on the procedure for showing the record and getting the key. This will be where every other procedure in her program will eventually return.

A procedure to get a one-letter command

```
edit ↵ comget ↵
```

1 Use `edit` to create a new procedure called `comget`.

```
while 1 ↵ print at 12,5;"Command?:"; ↵ let c$=getkey() ↵ print c$
↵ endwhile ↵ ↵ ESC
```

2 Type the following lines:

```
while 1
print at 12,5; "Command ?: ";
```

Note the semicolon at the end of the `print` statement, and the one used to separate the two print items. The print item `at` can be used to position the cursor directly with `print` as well as `input` commands.

```
let c$=getkey()
```

The line `let c$=getkey()` waits for a key to be pressed, then stores that character in the variable `c$`.

The variable name `c$` has been chosen to stand for command, and is short to save on typing. It also saves space in the procedures.

```
print c$
```

Unlike `input`, `getkey()` does not display the key pressed. It is useful to see it in case you press the wrong key and wonder why the program isn't working.

`endwhile` 4 Finish the procedure by closing the `while` loop.

↵ 5 Press **ENTER** to leave insert mode and then

ESC 6 leave `edit`.

The GETKEY() function

The function `getkey()` causes a procedure to stop and wait for a single key to be pressed. The function returns the value of the key pressed as a character. That is, it returns the character 'A' when **SHIFT** and `A` are pressed, and '9' (as a string) when 9 is pressed.

Unlike `input` the **ENTER** key does not have to be pressed afterwards, and the character does not appear on the screen unless `print` is used.

`print getkey()` ↵

1 Type `print getkey()` and press **ENTER**
The cursor in the WORK AREA waits on top of the prompt. The command will not complete itself until you press a key.

□ (*not* ESC) 2 Press any letter or number key. The letter or number is immediately printed on the bottom line of the display, and ARCHIVE is ready for another command.

Do not press **ESC** when using `getkey()`, or any procedure using it will be cancelled. Any other key, however, will cause the procedure to continue.

Running the procedure COMGET

`trace` ↵ 1 Switch on the program trace

`comget` ↵ 2 Type `comget` and press **ENTER**
The message '`Command ? :`' appears on the 12th line, 5 columns in.

□ (*not* ESC) 3 Press any key except ESC
The character appears after the message, and the cursor appears over it.

Pressing a key causes the character to appear immediately after the prompt message.

Pressing another key will cause that character to appear in the same place.

Note that the cursor in the DISPLAY AREA does not follow the prompt message. It does not indicate where the next character will appear when `getkey()` is used.

ESC 4 Press ESC to stop the procedure and then

`trace` ↵ 5 turn `trace` off.

Obviously, the procedure is useless as it stands, and another procedure is needed to take action according to the key pressed.

Naming procedures

The procedure names used in this chapter might seem a little odd, but are used as a compromise between several requirements:

- Procedures which begin with the same letters appear together in `edit`, and procedures which deal with the same task can be easily cross-referred.
- Procedure names should remind you of what the procedure does.
- Procedure names should be as short as is convenient, to save space in memory.

A procedure to act on keys pressed

`edit` ↵ F3 `n comcheck` ↵

1 Create a new procedure called `comcheck`.

```
if c$="S":stop:endif ↵ if c$="N":next:endif ↵ if c$="B":back:endif
↵ if c$="A":alter:endif ↵ if c$="I":insert:endif ↵ ESC
```

2 Insert the following line:

```
if c$="S":stop:endif
```

If the key s is pressed, the procedures will stop.

```
if c$="N":next:endif
if c$="B":back:endif
```

If the keys n or b are pressed, `next` or `back` are performed. As soon as they finish, the loop in `comget` starts again, and ARCHIVE waits for another keypress.

```
if c$="A":alter:endif
if c$="I":insert:endif
```

The commands `alter` and `insert`, called by pressing `a` and `i`, allow records to be amended exactly as if the commands were issued at the keyboard. The full command names are used here (rather than the short names in the `easy` procedures) because they work faster. The FUNCTION KEYS normally used to save or abandon records work as before. The CONTROL AREA still changes its prompts.

If any letter apart from these is pressed, the loop will take the procedure back to the `getkey()` command again.

The procedure should look like this:

```
proc comcheck
  if c$="S": stop : endif
  if c$="N": next: endif
  if c$="B": back: endif
  if c$="A": alter: endif
  if c$="I": insert: endif
  endproc
```

ESC `save "wedcom"` ↵

3 Leave `edit` and save the procedures in a program file called `wedcom`. This stands for 'wedding commands' as distinct from `wedfiles`, which contains procedures for the file handling tasks.

Amending the COMGET procedure

Many of these commands either move onto a new current record, or change one. It is therefore useful to show the current record every time ARCHIVE comes back from `comcheck` and goes round the loop in `comget` to get another command.

Sometimes the current record will not be different, but a superfluous `sprint` is preferable to dotting them about various procedures, and possibly forgetting to use it altogether.

Include `sprint` in `comget`:

edit ↵ TAB 1 Edit `comget`

↓ F4 `sprint` ↵ 2 Add `sprint` after the `while` line.
This causes the display to show the current record whenever the loop starts.

Remember that the record display does not work automatically while a procedure is running. The `sprint` also forces the last record added to be displayed properly when `insert` finishes.

↓ ↓ F5 SHIFT → → (3 TIMES) `upper(` ↓ `)` ↵

3 Change the `let` statement to read:

```
let c$=upper(getkey())
```

This converts any lower case letter key-press into upper case. Any following reference to `c$` can safely assume it will not be lower case.

↑ F4 `comcheck` ↵ ESC

4 Insert the line `comcheck` before the `endwhile`.

The procedure now looks like this:

```
proc comget
  while 1
    sprint
    print at 12,5; "Command ?: ";
    let c$=upper(getkey())
    print c$
    comcheck
    endwhile
  endproc
```

ESC 5 Leave `edit` and

`merge "easy"` ↵ 6 merge the `"easy"` procedures, as some of them will be called by `comcheck`.

`edit` ↵ TAB (4 TIMES) F3 `d` ↵ F3 `d` ↵ ESC

7 Delete the procedures `d` and `e` as they are not needed in this procedure and leave `edit`.

Menus

Procedures and programs which offer a set of choices that can be selected by a single key press are often called *menus*. Like a restaurant menu, you choose what you want from what's available.

Computer menus act like a platform from which all other options begin and to which they return. They are a useful and stable point of reference.

Using COMGET with the `guests` file

`display` ↵ 1 Display the current record of the `guests` file.

`comget` ↵ 2 Type `comget` and press **ENTER**
The '`Command ?:`' prompt appears.

`n` 3 Press `n`
The letter appears after the prompt, and the next record appears.

`n n` 4 Press `n` twice, very quickly
The next record appears, but not the one after.

Any keys pressed since the last command and not yet acted on are disregarded by `getkey()`, which starts by clearing the keyboard *buffer*.

This means you have to wait until a command finishes and `getkey()` is ready before pressing another key.

This is not easy to do at the moment, because the last letter pressed remains on the screen after the prompt. It is not easy to tell when the procedure is ready to get the next key stroke.

s 5 Press s to stop the procedure.

Providing a cursor for GETKEY()

Alter `comget` so that the old character is cleared when its command is over.

`edit` ↵ TAB (3 TIMES)

1 Use `edit` and select the `comget` procedure.

↓ (3 TIMES) F5 ↓ `paper 6;" "` ↵

2 Add a white space to the prompt message in the first `print` statement, so that it looks like this:

```
print at 12,5;"Command ?: ";paper 6;" "
```

This acts as a cursor to show that a command is expected. It will be overprinted by the next command key pressed.

↓ ↓ F5 SHIFT ↓ `at 12,16;` ↵ ESC

3 Change the `print c$` line to read:

```
print at 12,16;c$
```

Characters always print on the screen after the last character printed. This means that the key press stored in `c$` will print after (rather than over) the new 'cursor' unless told not to do so (by using the `at` print item).

Starting with the '`C`' of `Command` as 5, the 'cursor' is the 16th character, so the 'got' key will replace it.

If it doesn't, you probably haven't copied the example prompt message exactly. Don't worry, merely adjust the `at` column to fit your own prompt.

The procedure now looks like this:

```
proc comget
  while 1
    sprint
    print at 12,5; "Command ?: "; paper 6; " "
    let c$=upper(getkey())
    print at 12,16; c$
    comcheck
    endwhile
  endproc
```

ESC 4 Leave edit.

comget ↵ 5 Try comget again
This time, as soon as a command is finished, the white space re-appears.

When the white space is present, press a command letter. If not, wait for the last command to finish.

Preventing errors from stopping a program

i 1 Press i
The display shows an empty record.

ESC 2 Press ESC to abandon insert
Not only does insert finish, but both the comget and comcheck procedures collapse.

edit ↵ TAB TAB

3 Use edit to change comcheck.

↓ (5 TIMES) F5 SHIFT↑ SHIFT↑ CTRL↓ error i:endif ↵

4 Change the command insert so the line reads: if c$="I":error i:endif

The procedure now looks like this:

```
proc comcheck
  if c$="S": stop : endif
  if c$="N": next : endif
  if c$="B": back : endif
  if c$="A": alter : endif
  if c$="I": error i: endif
  endproc
```

ESC 5 Leave edit

comget ↵ 6 Try comget again

i 7 Press i

ESC 8 Press ESC to leave insert
This time the procedure does not fail.

Note that it redisplays the current record.

Note: In version 1 of ARCHIVE, pressing the ESC key is the only way of stopping insert, F4 cannot be used. The F4 key does not cause procedures to collapse.

s 9 Press s to stop the procedure

save ↵ wedcom1 ↵

10 save the procedures, to the program file wedcom1.

It now contains the procedures a, b, comcheck, comget, i, n.

Flow charts

Every teacher of programming tells you that you write better programs and procedures if you analyse the 'flow' of the program using a *flow chart*. A flow chart is a diagram using shapes and arrows to show a program's steps and connections. Almost every programmer tells you that he can't be bothered to use them, but knows he would work faster, with fewer mistakes, if he did.

There is no room here to discuss the rules and methods of constructing a flow chart, but essentially it is very similar to Miranda's step-by-step 'plain English' analysis. Its purpose is to make you consider each step in turn.

If the idea appeals to you, find a book that explains the method fully.

The ERROR command

The command `error` absorbs any errors in the procedure which follows it. If the procedure collapses for any reason which would normally result in an error or `<ESC>` message, that procedure will stop, but no error message will be displayed, and the calling procedure will not collapse.

The `error` command cushions the procedure it is used in from the effects of an error in the procedures it calls.

In the above example, if the procedure `i` collapses because the command `insert` fails, the calling procedure (with the line `error i` in it) will not collapse.

`error` only works with procedures, however, so it cannot be used with ARCHIVE's own commands. This is why the `easy` procedure `i` is used here. The line `error insert` is wrong and would itself cause an error.

A procedure to close all open files

`edit` ↵ F3 `n closeall` ↵ `while 1` ↵ `close` ↵ `endwhile` ↵ ESC

1 Create the following procedure:

```
proc closeall
  while 1
    close
    endwhile
  endproc
```

ESC 2 Leave `edit`.

Running CLOSEALL

```
closeall ↵
```

1 Type `closeall` and press **ENTER**. The `guests` file is closed and saved back onto tape, and the error message `Error 94: file not open` is displayed. The procedure automatically fails as soon as there are no more files to close.

The procedure continues to loop and `close` any open files until there are no more, when the procedure stops because of the error.

The ERRNUM() function

```
error closeall ↵
```

1 Type `error closeall` and press **ENTER**
There are no files to close, so the procedure immediately fails, but does not cause an error message. If the line `error closeall` had been used in a procedure, that procedure would have continued uninterrupted.

```
print errnum() ↵
```

2 Type `print errnum()` and press **ENTER**
The number 94 appears in the DISPLAY AREA. This is the number of the error which caused `closeall` to stop, although no error message was displayed.

The function `errnum()` returns the number of the last error, or 0 if a procedure used with the command `error` works (ie. does not produce an error).

The ERRNUM() function

Whenever an error occurs, the function `errnum()` returns the number of the error.

Normally, errors in commands or procedures cause them to collapse immediately and to print an error message and number in the WORK AREA. The command `error` changes this.

`errnum()` is primarily used to check for errors after calling a procedure with the command `error`. If `errnum()` is not used afterwards, any errors will be trapped and may go unnoticed. There is no indication that an error has taken place when it is trapped by `error`.

The function `errnum()` returns the value 0 if no error is encountered in procedures called by `error`.

The **ESC** key causes `errnum()` to return 27. Note that this is the same number as the ASCII code for *escape*.

Error reporting

It is not a good idea to indiscriminately use `error` to trap errors, as they do perform a useful function.

For example, if you use `alter` with a file which was opened with `look`, you will not know until you press F5 that ARCHIVE refuses to change the record. This is because ARCHIVE will not allow you to *write* to a *read only* file, like one opened with `look`.

If the command has been used in a procedure called with `error`, you might not notice at all that the `alter` has had no effect.

Similarly, `closeall` might fail to completely save an open data file back to tape, and give no warning.

The nature of an error can be checked using the `if` command, and an appropriate message displayed, or action taken, eg:

`edit` ↵ F3 `n errchk` ↵ `if errnum()>0` ↵ `print at 14,5;ink 2;paper 6;"Error";errnum()` ↵ `endif` ↵ ESC

1 Create this procedure:

```
proc errchk
  if errnum()>0
    print at 14,5; ink 2;
      paper 6; "Error "; errnum()
    endif
  endproc
```

SHIFT TAB SHIFT TAB ←(4 TIMES) F5 SHIFT↓ SHIFT↓ CTRL← `error a: errchk: endif` ↵

2 Add the command `error`, and the word `errchk` to the line which calls the procedure `a` in `comcheck`.

← F5 SHIFT↓ (4 TIMES) `errchk:` ↵

3 Insert the word `errchk` into the line which calls `i`

These lines now look like this:

```
if c$="A": error a: errchk: endif
if c$="I": error i: errchk: endif
```

ESC 4 Leave `edit`.

Using ERROR and ERRNUM()

- Always use `error` and `errnum()` together.
- First call a procedure using `error`, then, when that procedure finishes, check the resulting `errnum()`.
- Remember that it is always (ultimately) an ARCHIVE command which causes the error, but that you can only `error` the procedure containing the command, and not the command itself.

- An error causes the procedure it occurs in to collapse. This collapses the procedure which called it, which in turn collapses the procedure which called it, and so on.

  This continues until the command line itself is reached, or until a procedure is reached which called one of the collapsing procedures with an `error` command. This traps the error, and prevents the collapse continuing.

The following examples show the effect of using `error` in different places in three procedures which call each other in turn and finally call a procedure with a mistake in it.

```
Example 1:
proc1           proc2           proc3           proc4
  proc2           proc3           proc4           prit "Hello"
  print "1"      print "2"      print "3"      print "4"
  endproc         endproc         endproc         endproc
```

Typing `proc1` on the command line will cause the error message `Error 1: Command not recognized`. This is because `print` is mispelled as `prit` in `proc4`.

```
Example 2:
proc1           proc2           proc3           proc4
  proc2           proc3           error proc4     prit "Hello"
  print "1"      print "2"      print "3"      print "4"
  endproc         endproc         endproc         endproc
```

Typing `proc1` on the command line will cause the error to be trapped in `proc3`. Only `proc4` will collapse. The DISPLAY AREA will show:

```
3
2
1
```

```
Example 3:
proc1           proc2           proc3           proc4
  error proc2     proc3           proc4           prit "Hello"
  print "1"      print "2"      print "3"      print "4"
  endproc         endproc         endproc         endproc
```

Typing `proc1` on the command line will cause the error to be trapped in `proc1`. The procedures 2,3 and 4 will collapse. The DISPLAY AREA shows only 1.

Note: any commands following a failed procedure or command are *not* executed. ARCHIVE returns immediately to the calling procedure.

Creating a new starting procedure

Miranda wants to create a starting procedure which opens the `guests` file and displays the first record.

This will supersede for most purposes the very basic procedures she created earlier (see Chapter 4). She includes all the commands that only need to be used once at the start of the program.

```
edit ↵ F3 n start ↵cls ↵print at 5,10;ink 2;paper 6;"Opening Files
... " ↵error closeall ↵open "guests" logical "g" ↵display ↵comget
↵ ESC
```

1 Use `edit` to create the procedure `start`.

```
cls
```

2 Type `cls` to clear the screen ready for the new program.

```
print at 5,10; ink 2;paper 6; " Opening Files .... "
```

3 It is always helpful to let whoever is using a program know what is going on. The `Opening Files` message also helps allay any worries about how long the Microdrives may be taking to do their job. Note the use of colour to highlight the message.

```
error closeall
```

4 Type `error closeall`
This closes any files which may have been accidentally left open.

```
open "guests"logical "g" ↵ display
```

5 Type `open "guests"logical "g" display`
`display` is used here because it only needs to be used once.

Remember that once the file has been `displayed`, `sprint` shows the current record, and `screen` redisplays it if it is cleared.

```
comget
```

6 Type `comget`
This calls the *menu* program, from now on the root of all the other procedures. If the procedures collapse, Miranda can restart them using `comget` rather than `start` which wastes a lot of time using the Microdrives, opening and closing files.

The procedure now looks like this:

```
proc start
  cls
  print at 5,10; ink 2;paper 6;
    " Opening Files ... "
  error closeall
```

```
open "guests"logical "g"
display
comget
endproc
```

The core of this procedure is the commands `open`, `display` and `comget`, but the other lines have important functions.

Remember that the command `run` looks for a procedure called `start` in the programs it loads.

Identifying programs and procedures

F3 n a1 ↵ rem WEDCOM_PRG 5 April 1984 ↵ ESC

1 Create the procedure `a1`, with the date and name of the program file in it. Use the command `rem` at the beginning of the line, to tell ARCHIVE not to treat what follows on the same line as a command, but to ignore it.

The procedure should look like this:

```
proc a1
  rem              WEDCOM_PRG    5 Apr 84
  endproc
```

This identifies the program, and the version date.

The procedure is called `a1` so it will appear near the top of the list of procedures in `edit`, and is therefore easy to refer to.

Add some other remarks to remind you which procedures are linked, and what they do.

```
REM comget: menu, sprints, getkey command c$
REM comcheck: called by comget, uses if to test c$
REM i: called by comcheck, (error) insert
REM errchk: prints errnum() message
(etc)
```

You may well feel right now that you do not need reminding how your own procedures work. You may soon forget however as time passes and you write procedures in new ways.

This procedure guide or index will also help anyone who needs to use or fix your procedures if you aren't available to explain them and how they relate.

Reading a set of remarks is easier than working through all the procedures.

Remarks are often written in the procedures they refer to, but this slows down the procedures considerably, especially if they are loops.

Note: when there are lots of procedures in a program, this procedure (`a1`), like the HELP procedure, becomes extremely large, and wastes space in the temporary memory.

The solution is to keep two versions of your final program on tape. One is fully annotated (with, for example, the file extension `_rem`) and the other is the compact, working program.

`ESC` 2 Leave `edit`

```
save "wedcom2" ↵
```

3 `save` the procedures as `wedcom2`.

This program contains the procedures `a1`, `start`, `comget`, `comcheck`, `closeall`, `errchk` and the `easy` procedures, `a`, `b`, `i` and `n`.

FINDING THE RIGHT PEOPLE

Miranda now goes on to create a procedure to find people in the guest list. Before she can change a record, she must find it, and `next` is a slow way of doing it.

```
merge ↵seemore ↵
```

1 `Merge` the programs file `seemore`

```
edit ↵
```

2 Use `edit`

`TAB` (8 TIMES) `F3` `d` ↵ `TAB` `F3` `d` ↵

3 Delete the procedures `listmemos` and `prinrec` as they will not be needed.

`F3` `n f` 4 Create a procedure called `f`, (short for 'find')

```
input at 13,5; ink 4; "Find ?: "; txt$
```

5 This prompts for the letters to be found, and stores them in `txt$`.

```
if txt$>"" ↵
```

6 If only the ENTER key is pressed, the variable `txt$` will be a null string, ie. it will have nothing in it. In this case, the procedure assumes you do not want to find any records and returns control to `comget`.

```
find txt$ ↵if found() ↵
```

7 `if found()` is a second 'if' clause (completely inside the first) which checks the result of the `find`.

```
sprint ↵seemore ↵
```

8 This calls the procedure which uses `continue` while records still match, as the first find may not be right.

```
else ↵
```

9 The `else` command is part of the `if` clause it follows, and provides an alternative course of action for occasions when the `if` expression fails. In this example, this happens when `found()` returns 0, ie. is not true.

```
print at 14,5; paper 6; ink 2; txt$+" not found" ↵
```

10 This prints a message if nothing is found.

```
endif ↵endif ↵
```

11 Finish the procedure by closing the two `if` clauses.

The procedure should look like this:

```
proc f
  input at 13,5; ink 4; "Find ?: "; txt$
  if txt$>""
    find txt$
    if found()
      sprint
      seemore
      else
      print at 14,5; paper 6; ink 2; txt$+" not found"
      endif
    endif
  endproc
```

Nested IFs

Notice that one `if` clause has been used inside another one. The second `if` clause will only be reached if the first is true. In computing jargon this is called *nested ifs*. `if` clauses can be nested to any depth, that is, you can have as many `if`s within `if`s within `if`s as you wish.

There is a practical limit however, as ARCHIVE's indentation of `if` clauses will finally force the lines off the edge of the `edit` screen.

Make sure that you position the corresponding `endif`s correctly when using several if clauses together. Remember to check that `endproc` is indented two spaces. This shows that the correct number of matching `end...` commands are present.

The ELSE Command

The word `else` is really part of the `if, endif` statement. Any commands following the `else`, up to `endif`, will be executed if the expression accompanying `if` is false.

Its use can be paraphrased as: '`if` this is true, do these commands, `else` (ie. otherwise) do the commands up to `endif`.'

Adding new options to COMCHECK

SHIFT TAB (3 TIMES) ↓ F4 `if c$="F":f:endif` ↵ ESC

12 Add the following as one of the `if` statements in `comcheck`:

```
if c$="F":f:endif
```

The procedure now looks like this:

```
proc comcheck
  if c$="S": stop : endif
  if c$="F": f: endif
  if c$="N": next: endif
  if c$="B": back: endif
  if c$="A": error a: errchk: endif
  if c$="I": error i: errchk: endif
  endproc
```

ESC 13 Leave `edit` and try out the new commands.

Altering the SEEMORE procedure

The procedure `seemore` as originally created goes on using the command `continue` until there are no more matches.

The only way to stop on the record you want is to press ESC, which is a little hit and miss if the display is too fast to read.

Change it to wait for a key press before continuing after a find.

`edit` ↵ TAB (10 TIMES) ↓ F4 `print at 13,5;ink 4;"Press any key to see more, or S to stop"` ↵ `if upper(getkey())+"S"` ↵ `return` ↵ `else` ↵ ESC

1 Add the following lines to `seemore` after the `while` command:

```
print at 13,5;ink 4;"Press any key to see
  more, or S to stop"
if upper(getkey())="S"
return
```

This command returns control immediately to `f`, ignoring all the lines following it.

`else` 2 `else`
If any key other than "s" is pressed, the commands after `else` are executed, and `seemore` continues to loop.

↓ ↓ F4 `endif` ↵ ESC

3 Add an `endif` before the `endwhile`.

↓ F4 `print at 14,5;paper 6;ink 2;"No more  "+txt$` ↵ ESC

4 Add a warning line at the bottom of the procedure to show that there are no more records to be found.

The procedure now reads:

```
proc seemore
  while found()
    print at 13,5; ink 4; "Press any key to see more, or S to stop"
    if upper(getkey())="S"
      return
      else
      continue
      sprint
      endif
    endwhile
  print at 14,5; paper 6; ink 2; " No more "+txt$
  endproc
```

ESC 5 Leave `edit`.

The RETURN command

The command `return` causes the procedure in which it is used to stop immediately, and to return control to the procedure which called it. A procedure normally only returns control when it has finished.

If the procedure was called from the command line, the > prompt reappears, ready for another command.

`return` is not like `stop`, which causes the whole program and all its running procedures to stop.

Using the program to find some names

`start` ↵ 1 Use the command `start` to set the program running.

`f` 2 Press `f`
The message '`Find ?:`' appears on the line below and the cursor is waiting after it.

`pew` ↵ 3 Type `pew` and press **ENTER**
The screen changes to show `Mabel Pew` and the message '`Press any key to see more, or S to stop`' appears.

(any key except `S`) 4 Press any key except `s`
The message '`No more Pew`' appears, because there are no more records with the letters `pew` in them.

`f` 5 Press `f` again

`fred` ↵ 6 Type `fred` and press **ENTER**
The message '`fred not found`' appears since no record contains the letters `fred`.

s 7 Press s to stop the procedure.

A procedure to clear unnecessary lines from the screen

There are now so many messages on the screen that Miranda is finding it hard to tell what is going on and what she is supposed to do next. She decides to add a procedure to clear lines from the screen which are no longer relevant.

```
edit ↵ F3 n blank;x,y ↵
```

1 Use `edit` to create a procedure for clearing a line. The procedure is called `blank` and sets up two parameters, `x` and `y`:

```
proc blank;x,y
```

This procedure needs to be passed two numeric parameters which determine which line should be cleared, starting at which column.

The procedure name is divided from the parameters `x` and `y` by a semi-colon, and they are divided from each other by a comma.

```
print at x,y;rept(" ",64-y+1);
```

2 Note the semicolon at the end of the print statement. Without it the procedure would cause the screen to scroll if used to clear the bottom line of the DISPLAY AREA.

The function `rept()` here repeats the literal space (" "), and is used with `print` to print spaces over the rest of the line, ie. `(64-y+1)` spaces.

↵ 3 Leave `insert`.

The procedure looks like this:

```
proc blank;x,y
   print at x,y;rept(" ",64-y+1);
   endproc
```

The function REPT()

The function `rept()` contains two pieces of information inside the brackets, divided by a comma. The first is a character or characters to be repeated, and the second is the number of times they are to be repeated.

For example, `print rept("x",20)` displays:
`xxxxxxxxxxxxxxxxxxxx`

The variables `x` and `y` are used to position the cursor at row `x` and column `y`. Thus the `rept()` function prints 64 (the width of the screen display) spaces less the number of columns before the printing starts. If printing starts at column 5, then 5-1, or 4, spaces procede it. Thus, 64-4, or 60, columns are cleared.

In other words, when passed two numbers as parameters, `blank` clears the line `x` from column `y` to the end. If you wish to use `blank` with a different display width, eg. 40, change the `64` to `40`.

Inserting BLANK into other procedures

TAB (3 TIMES) 1 TAB down to the procedure `comget`.

↓ ↓ F4 `blank;13,5` ↵ ESC

2 Insert a line just after the `sprint` line of the procedure `comget`:

```
blank;13,5
```

This will clear from column 5 to the end of the line every time a command finishes, whether it used line 13 or not.

Error messages also need to be cleared after a new command is fetched in `comget`.

↓ ↓ F4 `blank;14,5` ↵ ESC

3 Insert this line after the `getkey()` line in the procedure `comget`:

```
blank;14,5
```

The procedure now looks like this:

```
proc comget
  while 1
    sprint
    blank;13,5
    print at 12,5; "Command ?: "; paper 6; " "
    let c$=upper(getkey())
    blank;14,5
    print at 12,16; c$
    comcheck
    endwhile
  endproc
```

Notice that prompt messages needed while a command (eg. `f` or `seemore`) is running appear on line 13, and warning or error messages appear on line 14.

Prompt messages need to be cleared as soon as the command finishes, and before the next key is pressed. The error messages, on the other hand, need to remain until after the next key is pressed, so that they stay on the screen long enough to be read.

ESC 4 Leave `edit`

save ↵wedcom3 ↵

5 save the procedures to "wedcom3"
"wedcom3" now contains the procedures:

```
a                f
b                i
blank            n
closeall         seemore
comcheck         start
comget           zoom
errchk
```

comget ↵ 6 Type comget and press **ENTER**

f 7 Press f

↵ 8 When prompted to input characters to be found, simply press **ENTER**

The Find prompt clears and the procedure waits for a new command key.

s 9 Leave comget by pressing S



The bounteous Blakes

Standard prompts and warnings

Miranda realizes that she will probably need prompt and warning messages in many other procedures, and typing the lines in full each time wastes space and time.

Also, if she decides to change the position of the messages on the screen, or their colour, she will have to change every prompt and message line in every procedure.

She creates two procedures to position and colour the prompts and warnings. They will be passed the relevant message as a parameter each time they are used.

`edit` ↵ F3 `n prompt;msg$` ↵ `print at 13,5;ink 4;msg$;` ↵ ESC

1 Create the procedure `prompt` as follows:

```
proc prompt;msg$
  print at 13,5;ink 4;msg$;
  endproc
```

F3 `n warning;msg$` ↵ `print at 14,5;paper 6;ink 2;" ";msg$;" "` ↵ ESC

2 Create the procedure `warning;msg$` as follows:

```
proc warning;msg$
  print at 14,5;paper 6;ink 2;" ";msg$;" "
  endproc
```

Note the two literal spaces either side of the message. This improves the legibility of the message when printed on a white background.

SHIFT TAB SHIFT TAB ↓ ↓ F5 CTRL ↓ (20 TIMES) `prompt` ↵ ↓ (8 TIMES) F5 SHIFT ↓ (6 TIMES) ↓ CTRL ↑ `warning` ↵

3 Adapt `seemore` to use these two new procedures.

The line:

```
print at 13,5;ink 4;"Press any key to see more, or S
to stop"
```

becomes:

```
prompt;"Press any key to see more, or S to stop"
```

The line:

```
print at 14,5;paper 6;ink2;" No more "+txt$
```

becomes:

```
warning;" No more "+txt$
```

The procedure now looks like this:

```
proc seemore
  while found()
    prompt; "Press any key to see more,
      or S to stop"
    if upper(getkey())="S"
      return
      else
      continue
      sprint
      endif
    endwhile
  warning; " No more "+txt$
  endproc
```

SHIFT TAB (4 TIMES) ↓ F5 SHIFT ↓ (4 TIMES) ↓ CTRL ↑ prompt SHIFT ↓ SHIFT ↓ ↓ CTRL → ↵ F4 input txt$ ↵ ESC → (7 TIMES) F5 SHIFT ↓ (6 TIMES) ↓ CTRL ↑ warning ↵

4 Adapt the procedure f, so that the line:

```
input at 13,5;ink 4;"Find ?: ";txt$
```

becomes:

```
prompt;"Find ?: "
```

and then add the line `input txt$`

Change the line:

```
print at 14,5;paper 6;ink 2;txt$+" not found"
```

so that it becomes:

```
warning;txt$+" not found"
```

ESC 5 Leave edit

The procedure now looks like this:

```
proc f
  prompt; "Find ?: "
  input txt$
  if txt$>""
    find txt$
    if found()
      sprint
      seemore
      else
      warning; txt$+" not found"
      endif
    endif
  endproc
```

Converting numbers to strings

Miranda finds that she has a slight problem with the `errchk` procedure. The function `errnum()` returns a numeric value. She wants to use the `warning` procedure to report the error number, but it only accepts strings as a parameter.

She first has to convert the error number to a string. This is done using the `str()` function.

The STR() function

`edit` ↵ TAB (7 TIMES) ↓ F5 ↓ `and errnum()<>27` ↵ ↓ F5 SHIFT ↑ (6 TIMES) ↑ CTRL ← `warning` SHIFT ↑ ↑ ↑ `+STR(` ↓ ↑ `,3,0)` ↵

1 Adapt `errchk` so that the line:

```
if errnum()>0
```

becomes:

```
if errnum()>0 and errnum()<>27
```

and the line:

```
print at 14,5;ink 2;paper 6;"Error ";errnum()
```

becomes:

```
warning;"Error ";+STR(errnum(),3,0)
```

ESC 2 Leave `edit`.

The procedure now looks like this:

```
proc errchk
  if errnum()>0 and errnum()<>27
    warning;"Error ";+STR(errnum(),3,0)
    endif
  endproc
```

The function `str()` requires three values to appear inside the brackets, separated by commas. They must all be numeric expressions.

Values passed to functions which define the way the functions work are called *arguments* in computer jargon. They are rather like parameters passed to procedures.

With `str()`, only the first one is the numeric value to be converted to a string.

The other two control the way the number is represented (eg. whether a whole number, or with decimal fractions), and the number of decimal places to be shown, if required.

All three arguments must be present, whether they have a purpose or not.

In this case, Miranda uses 3 as the second argument, denoting 'general format' representation, ie. as it comes.

She uses 0 as the third argument, determining decimal places, as none are required with the 'general format' representation, (which shows them if they exist, or not if they don't). This is called a 'dummy' argument.

The arguments available for use with `str()`, and what they mean, are more fully explained at the end of Chapter 12 and in the *Key Words* section at the back of this book.

Using the number-logging procedures

Miranda now has enough command options available to start updating the records of guests who have replied. It would be simpler, however, if (instead of pressing 'a' for `alter` and skipping 5 fields to reach `partyof` each time) she could simply enter the number directly into the `partyof` field.

`edit` ↵ F3 `n coming` ↵ `prompt;"Number coming?:"` ↵ `input partyof` ↵ `let coming$="y"` ↵ `update` ↵ ESC

1 Use `edit` to create the procedure `coming` as follows:

```
proc coming
  prompt;"Number coming ?: "
  input partyof
  let coming$="y"
  update
  endproc
```

This procedure will wait for a number to be entered, and then update the partyof field of the current record with that number, and simultaneously put an 'y' for `yes` in the `coming$` field.

SHIFT TAB SHIFT TAB ↵ F4 `if c$="C":coming:endif` ↵ ESC

2 Add another option to `comcheck`:

```
if c$="C":coming:endif
```

The procedure now looks like this:

```
proc comcheck
  if c$="S": stop : endif
  if c$="C": coming: endif
  if c$="F": f: endif
  if c$="N": next: endif
  if c$="B": back: endif
  if c$="A": error a: endif
  if c$="I": error i: endif
  endproc
```

ESC 3 Leave `edit`.

Logging the first reply

Miranda has a reply from Bill Hackney, drama critic of the Marston Gazette, who is delighted to be able to come to the wedding. To run the program:

`comget` ↵ 1 Type `comget` and press **ENTER**

`f` 2 Press `f`
The prompt appears.

`hack` ↵ 3 Type `hack` and press **ENTER**
The screen changes to show Bill Hackney's record.

`s` 4 Press `s` to prevent `seemore` from finding any more 'hack's if there happen to be any.

`c` 5 Press `c`

When prompted for the number coming:

`1` ↵ 6 type `1` and press **ENTER**
The display changes to show the new contents of the `coming$` field and `comget` is ready for another command key.

`s` 7 Press `s` to leave `comget`.

A procedure to help identify the command letters

Miranda is satisfied for the moment with the tasks that the program `wedcom` can perform, but decides to add a help screen to remind herself (and the twins if she can manage it) which keys to press.

`edit` ↵ TAB (5 TIMES) ↓ (4 TIMES) F5 ↓ `;paper 0;"(Press ? for help)"` ↵

1 Change `comget` so that the command prompt line now reads:

```
print at 12,5;"Command ?: ";paper 6;" ";paper 0;"
(Press ? for help)"
```

The procedure now looks like this:

```
proc comget
  while 1
    sprint
    blank; 13,5
    print at 12,5; "Command ?: "; paper 6; " "; paper 0; " (Press ? for help)"
    let c$=upper(getkey())
    blank; 14,5
    print at 12,16; c$
    comcheck
    endwhile
  endproc
```

SHIFT TAB F4 `if c$="?":help:endif` ↵ ESC

2 Add a new `if` clause in `comcheck` to find a procedure called `help` when a ? is pressed:

```
if c$="?":help:endif
```

The procedure now looks like this:

```
proc comcheck
  if c$="?": help: endif
  if c$="S": stop : endif
  if c$="C": coming: endif
  if c$="F": f: endif
  if c$="N": next: endif
  if c$="B": back: endif
  if c$="A": error a: endif
  if c$="I": error i: endif
  endproc
```

3 Create the new procedure `help` as follows:

```
proc help
  cls
  print "PRESS ONE OF THE FOLLOWING KEYS TO PERFORM A TASK"
  print "B  go back a record"
  print "N  go to next record"
  print "A  alter the record on the screen"
  print "I  insert a new record"
  print "F  find record(s)"
  print "C  log a reply, and numbers coming"
  print "S  stop"
  print
  print "Press any key to continue";getkey()
  screen
  endproc
```

4 Update the `a1` procedure to include the new procedures. Abbreviate the phrase `called by` to ().

ESC 5 Leave `edit`

`save` ↵ `wedcom` ↵ `save` ↵ `mdv1_wedcom` ↵

6 `save` the procedures to `wedcom`, and back them up to `mdv1_` as well, first having checked that you are using the correct backup tape by looking at the tape label.

7 Make an extra backup copy and label it 'for exercises'. You will need it in a few moments.

wedcom now contains the following procedures:

```
proc a
  errchk
  alter
  endproc
proc a1
  rem      WEDCOM_PRG  6 April 1984
  REM comget: menu, sprints, getkey command c$
  REM comcheck: (comget), uses if to test c$
  REM i: (comcheck), (error) insert
  REM errchk: prints errnum() message
  REM a: (comcheck), (error) alter
  REM blank: (comget), blanks rest of line at co-ordinate x,y
  REM closeall: (comget), closes all open files
  REM coming: (comcheck), updates partyof and send$ fields
  REM f: (comcheck), finds text, stored as txt$
  REM help: (comcheck), lists available options
  REM prompt: standard proc to print a prompt message (msg$) at 13,5
  REM seemore: (f): continues to find records until no more
  REM start: closes all files, opens and displays guests
  REM warning: standard proc to print warning message (msg$) at 14,5
  endproc
proc b
  back
  endproc
proc blank; x,y
  print at x,y; rept(" ",64-y+1);
  endproc
proc closeall
  while 1
    close
    endwhile
  endproc
proc comcheck
  if c$="?": help: endif
  if c$="S": stop : endif
  if c$="C": coming: endif
  if c$="F": f: endif
  if c$="N": n: endif
  if c$="B": b: endif
  if c$="A": error a: endif
  if c$="I": error i: endif
  endproc
proc comget
  while 1
    sprint
    blank;13,5
    print at 12,5; "Command ?: ";paper 6; " ";paper 0;" (Press ? for help)"
    let c$=upper(getkey())
    blank;14,5
    print at 12,16; c$
```

```
    comcheck
    endwhile
  endproc
proc coming
  prompt; "Number coming ?:"
  input partyof
  let coming$="y"
  update
  endproc
proc errchk
  if errnum()>0 and errnum()<>27
    warning; "Error "; +str(errnum(),3,0)
    endif
  endproc
proc f
  prompt; "Find ?: "
  input txt$
  if txt$>""
    find txt$
    if found()
      sprint
      seemore
      else
      warning; txt$+" not found"
      endif
    endif
  endproc
proc help
  cls
  print "PRESS ONE OF THE FOLLOWING KEYS TO PERFORM A TASK"
  print "B  go back a record"
  print "N  go to next record"
  print "A  alter the record on the screen"
  print "I  insert a new record"
  print "F  find record(s)"
  print "C  log a reply, and numbers coming"
  print "S  stop"
  print
  print "Press any key to continue"; getkey()
  screen
  endproc
proc i
  errchk
  insert
  endproc
proc n
  next
  endproc
proc prompt; msg$
  print at 13,5; ink 4; msg$;
  endproc
proc seemore
  while found()
    prompt; "Press any key to see more, or S to stop"
    if upper(getkey())="S"
```

```
        return
        else
        continue
        sprint
        endif
      endwhile
    warning; " No more "+txt$
    endproc
  proc start
    cls
    print at 5,10; ink 2; paper 6; " Opening Files ... "
    error closeall
    open "guests"logical "g"
    display
    comget
    endproc
  proc warning; msg$
    print at 14,5; paper 6; ink 2; " "; msg$; " "
    endproc
```

```
kill ↵wedcom1_prg ↵kill ↵wedcom2_prg ↵kill wedcom3_prg ↵
```

6 `kill` the early versions of the program, as they have been superseded.

```
comget ↵?
```

7 Try out the new help screen – type `comget` and press **ENTER**, then press ?

```
s
```

8 Press `s` to leave `comget`.

9 Remove your tape in mdv2_ and label it '*before doing exercises*'.

10 Insert the backup you have just made and labelled '`for exercises`'.

EXERCISES

The following exercises involve adding further tasks to the program `wedcom`. Each one can be called from the main `Command ?` prompt by pressing a single letter.

Most of them will be used in the next chapter to provide a full set of options on a sample business program.

Remember to add the new options to the `comcheck` and `help` procedures.

Remember that prompts and additional input appear on line 13, and warnings on line 14.

Suggested answers appear at the end of the exercises.

1 Command *n*

Adapt the procedure `n` so that if the end of the file is reached, the message `End of File` is displayed. Change the line in `comcheck` which calls `next` to call `n` instead.

2 Command b

Adapt the procedure `b` to produce the message '`Start of File`' when used on the first record. Change `comcheck`.

3 Command *z*

Add the procedure `zoom` to the `wedcom` program. Adapt it so that it can be stopped by pressing ESC without stopping the whole program. Remember the procedure `n`. Print a message on line 13 reminding the person using it that ESC is available. This way a record can be scanned for, and `zoom` stopped when it is seen.

4 Command *r*

Exactly like `zoom`, but reverses through the file.

5 Command *l* (one)

Go to the first record (the letter `f` has already been used).

6 Command *l* (el)

Go to the last record.

7 Command *g*

Go to a specified record number. This will require the number to be additionally input (compare `f` and `coming`).

8 Command *j*

Jump a specified number of records forwards or backwards from the current record. Input how many as either a positive or a negative number. Remember that numeric variables do not end in a `$`.

9 Command *d*

Delete a record. In case the key is pressed by accident, add an input line asking for confirmation that the record should in fact be deleted.

- Make sure you keep your answers on your tape labelled '*for exercises*'.

SUGGESTED ANSWERS

The following procedures are not necessarily the best, and certainly not the only solutions to the problems set at the end of Chapter 9.

Most of them will be used or referred to in the next chapter, so do read through them and add them exactly as described to your tape with `wedcom` that you labelled '*before doing exercises*'.

1 Command *n*

```
proc n
  next
  if eof()
    warning; "End of file"
    endif
  endproc
```

2 Command *b*

```
proc b
  back
  if recnum()=0
    warning; "Start of file"
    endif
  endproc
```

3 Command *z*

```
proc zoom
  escmsg
  while not eof()
    sprint
    n
    endwhile
  endproc
```

The procedure `n` is used rather than `next` because it provides a warning message when the end of the file is reached.

```
escmsg
  prompt; "Press ESC to stop"
  endproc
```

The procedure escmsg has been created because it can be used in other procedures, such as the next one.

Add a line to comcheck which reads:

```
if c$="Z":error zoom:endif
```

4 Command *r*

```
proc reverse
  escmsg
  while recnum()>0
    b
    sprint
    endwhile
  endproc
```

Add a line to comcheck which reads:

```
if c$="R":error reverse:endif
```

5 Command *l* (one)

Add a line to comcheck which reads:

```
if c$="1":first:endif
```

6 Command *l* (el)

Add a line to comcheck which reads:

```
if c$="L":last:endif
```

7 Command *g*

```
proc goto
  prompt; "Goto record no. ?"
  input num
  position num
  endproc
```

Notice that num is a numeric variable.

Add a line to comcheck which reads:

```
if c$="G":goto:endif
```

8 Command *j*

```
proc jump
  prompt; "Records to jump?"
  input num
  position recnum()+num
  endproc
```

Add a line to comcheck which reads:

```
if c$="J":jump:endif
```

The variable num is used here as in the previous procedure. It's function is purely temporary, and like txt$ for string input, keeping the same input name consistently prevents numerous superfluous variables from clogging the memory.

The command local can be used to the same effect.

Note that if a positive number is entered into num, the file will be positioned that many records forward from the current record. If a minus number is entered, the opposite happens, as a + and a - result in a -.

9 Command *dele*

```
proc dele
  prompt; "Confirm delete (y/n)"
  input txt$
  if lower(txt$)="y"
    delete
    endif
  endproc
```

Add a line to comcheck which reads:

```
if c$="D":dele:endif
```

HELP

The `help` procedure should now look something like this:

```
proc help
  cls
  print "PRESS ONE OF THE FOLLOWING KEYS TO PERFORM A TASK"
  print "B  go back a record"
  print "N  go to next record"
  print "A  alter the record on the screen"
  print "I  insert a new record"
  print "F  find record(s)"
  print "C  log a reply, and numbers coming"
  print "S  stop"
  print "D  delete current record"
  print "1  go to first record"
  print "L  go to last record"
  print "Z  zoom/fast forwards"
  print "R  reverse/fast backwards"
  print "J  jump a number of records forwards or backwards (minus)"
  print "G  go to a specific record number"
  print
  print "Press any key to continue";getkey()
  screen
  endproc
```

If the help screen gets too large for the display setting, remove headings or double up letters (eg. B/N back/next record)

SAVING YOUR WORK

Alter your answers to the exercises so that they are the same as the suggested answers, or simply key in the suggested answers, before continuing.

`save ↵wedcom ↵` 1 Save the procedures you now have in memory to `wedcom`.

`dir ↵ ↵` 2 Do a directory of drive 2 to check the date on the `_lab` file.

`dir ↵mdv1_ ↵` 3 Do a directory of your backup tape, in drive 1, to check the date on the `_lab` file on it to make sure it is the correct backup tape. If it is the same as your working tape, use your other backup tape.

`save ↵mdv1_wedcom ↵`

4 Save the same procedures to `mdv1_wedcom`.

`new ↵` 5 Clear all procedures and data files from temporary memory by typing `new` and pressing **ENTER**.

`backup ...` 6 Backup files created or changed since the tape was last used.

`save"Apr6_lab"` ↵ F5 SHIFT ◆ ◆`mdv1_` ↵

7 Save label files with today's date onto both the working tape and the backup tape in drive 1.

8 `kill` the old `_lab` files on both tapes.

SUMMARY

As the replies to the wedding invitations start to flood in, Miranda decides to write a set of procedures that will enable her to see who has replied, and how many people from that address are coming.

First of all, she creates two procedures, `comget` and `comcheck` which work together to get a one-letter command and then act in a certain way depending on which key has been pressed. The procedure `comget` uses the `getkey()` function and the variable `c$` to get a command from the keyboard. The `comcheck` procedure checks which key has been pressed, and acts accordingly. Miranda then saves these procedures with the program name `"wedcom"` (short for wedding commands). She then amends `comget` slightly by adding the line `sprint`, so that the current record is always displayed when it starts. She also uses the function `upper()` to convert any letter typed in at the keyboard in response to `getkey()` into upper case.

Eager to try out her new procedures Miranda uses `merge` to bring the `easy` procedures into current memory, as some of them are called by `comcheck`. She then tests them out on the `guests` file. She realises that they would be easier to use if `getkey()` had a cursor to indicate that ARCHIVE is waiting for a key to be pressed. She uses the `paper` command to provide a white cursor at the place where the command letter is entered.

She now goes on to use the `error` command to prevent the program from collapsing if there is an error (like pressing ESC in `insert`). She also uses this command with a new procedure she creates, called `closeall`, which closes all open data files. Using `error`, the procedure does not collapse when there are no open files left to close. She then uses the function `errnum()`, in a procedure called `errchk`, which prints the error number when the command `error` has been used, so that errors which might cause problems do not go by unremarked.

Now Miranda creates a procedure to find a particular person. By linking this procedure, `f`, with the `seemore` procedure, which she created in Chapter 7, ARCHIVE will continue to find names which contain the letters entered.

After testing her work, Miranda adds a procedure called `blank` to the program, which clears old message and warning lines from the screen.

She now creates two standard procedures to call prompt and warning messages which can be called by any other procedures.

She creates a procedure which, when called, lists the various options available at the command line, which she calls `help`.

Screen Design

The `display` command is fast and simple to use, but has several basic limitations:

- If a file has more fields than fit on the screen, the fields which are not seen cannot be edited or inserted using `alter` or `insert`.
- Two files cannot be `display`ed together on the same screen.
- The logical file name is `display`ed, not a full file name or heading.
- The field names (often abbreviated to an almost meaningless single word) are displayed rather than helpful prompts.

This chapter shows you how to design your own screen.

- The following ARCHIVE commands are introduced:

`sedit`	– edit a screen layout
`ssave`	– save a screen layout onto tape
`sload`	– load a screen layout from tape
`sinput`	– fetch input from a screen layout

Points to remember

- Custom-designed screen layouts behave exactly like the `display` layout.
- A screen created in `sedit`, like procedures in `edit`, must be saved onto tape, or it will be lost.
- You can edit the `display` layout if you like, but may have to clear it from `sedit` using F3 and `c`.
- Miranda's program assumes prompts and messages use lines 12 to 13.

DESIGNING YOUR OWN SCREEN

Miranda thinks that the standard record display layout provided by ARCHIVE is functional, but not very attractive. It does not allow fields to appear anywhere except on the left of the screen.

It is possible to design your own screen layout with a series of `input at` statements. These would only fetch one field at a time, however, and the `input` command does not show the current value of the field. It would also be slow.

Instead, Miranda decides to use the command `sedit` to create a screen layout which works in similar way to the `display` layout used with commands like `insert` and `amend`.

It will look like this:

```
.....................................................
                WEDDING GUEST LIST

Name
Address
                                          Send ?
                                          No. coming
                                          Staying ?

.....................................................
```

The SEDIT command

The word `sedit` is short for *screen editor*.

It allows you to create and edit screen layout. Unlike `edit`, which edits procedures one line at a time in the WORK AREA, `sedit` lets you to move the cursor and type text directly into the DISPLAY AREA.

The command `sedit` allows you to create a screen layout which behaves exactly like ARCHIVE's standard display.

Note: If the command `display` has been used, the standard display for the file will be in memory, and can be edited (changed) using `sedit`. Since you will probably want to start afresh, clear the standard display from the `sedit` screen using **F3** followed by `c` and **ENTER**.

Alternatively, you may find it useful to leave the standard display temporarily on the screen to remind you of the field names. They can then be erased using the **SPACE BAR**.

Creating a screen display using SEDIT

`sedit` ↵

1 Type `sedit` and press **ENTER**
The DISPLAY AREA clears, and the CONTROL AREA prompts you to use the cursors and **ENTER** to move about the screen.

You can now place prompts and messages for display and input on the screen, and tell ARCHIVE where to fetch variables and fields.

The result can be saved and used instead of `display` whenever you require.

F3 `c` ↵

2 Clear the standard display from the screen.

The screen title

◆ (20 TIMES)

1 Press ◆ until the cursor is about a third of the way along the top line.

F3 `p` SPACE ↵

2 Set the background colour for the screen title to red. To do this, select the `paper` option after pressing F3, by pressing `p`. The colour appears in the WORK AREA, and the `paper` number appears in the CONTROL AREA. Press the **SPACE BAR** until the word `red` appears in the WORK AREA, then press **ENTER** to tell ARCHIVE that you have selected that colour.

F3 `i` ↵

3 Set `ink` (the colour of the letters) to white in the same way that you set the `paper` colour to red.

CAPS LOCK `WEDDING GUEST LIST` ↵ CAPS LOCK

4 Type `WEDDING GUEST LIST` and press **ENTER**
The cursor moves to the beginning of the next line.

The colour of the background is red, and the letters are white.

The background and letters will continue to be these colours until you set different ones.

↵

5 Press **ENTER** again

Using colour in SEDIT

You can fill the whole screen layout with the colours black, green, white or red using the `paper` and `ink` options from the `sedit` F3 menu, if you choose.

Once set, `paper` and `ink` colour every prompt and variable you type until you set different ones. This is chronological rather than spatial – ie. you can set a colour and move anywhere on the screen, not just forwards.

If you press the **SPACE BAR**, a square with the `paper` colour appears. To move around the screen without changing the background colour, use the arrow keys.

You can set the colour of variables as well as text (not in version 1).

Setting the first variable

F3 p SPACE (4 TIMES) ↵

1 Set the paper colour back to black

F3 i SPACE (4 TIMES) ↵

2 Set the ink colour back to white

`Name` 3 Type `Name`

↓ (6 TIMES) 4 Press ↓ six times.

F3 5 Press F3
The CONTROL AREA changes to display four keys which may be pressed to perform a task. The use of F3 in `sedit` is very much like its use in `edit` in this way.

`v` 6 Press `v`
The prompts box in the CONTROL AREA changes again, and a prompt '`Variable name:` ' appears in the WORK AREA.

ARCHIVE is waiting for you to type in the name of a variable that will be fetched from the screen at this point, as if the command `input` were used.

`name$` 7 Type `name$`

↵ 8 Press **ENTER**
The prompt box changes again. ARCHIVE expects you to mark the area where the variable is to be displayed and input.

9 Press any key *except* **ENTER**
A dot appears in the DISPLAY AREA. Note that it is not the character of the key which you actually pressed. Pressing any key except **ENTER** will produce a dot.

□ (39 TIMES) 10 Press any key another 39 times

↵ 11 Press **ENTER**
The cursor does not go to a new line, but the CONTROL AREA changes back to the main layout mode prompts. The WORK AREA has cleared.

You have finished setting the variable position.

↵ 12 Press **ENTER**

The variable names in brackets do not appear on the screen; they are there to show you where each variable is set in relation to the others.

As you set the variables, dots appear on the screen to indicate how long that variable will be. These dots disappear when you leave `sedit`.

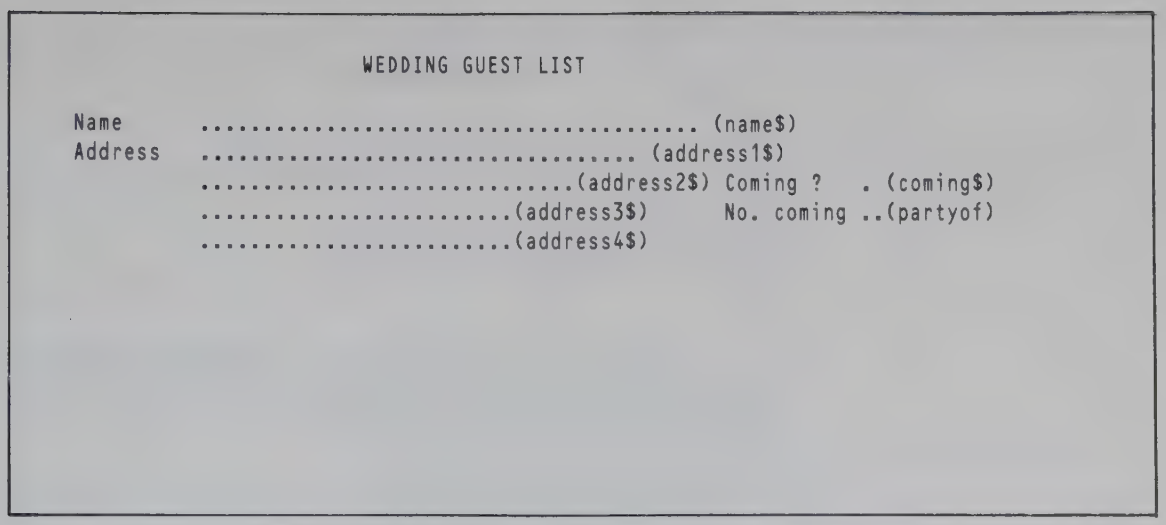


Figure 10.1 – Diagram of proposed screen design

Setting second variable

`Address`	1	Type `Address`
◆ ◆ ◆	2	Press ◆ three times to bring the cursor under the first dot.
F3	3	Press F3
v	4	Press v
`address1$` ↵	5	Type `address1$` and press ENTER Note that ARCHIVE does not check to see if the name is a valid variable name.
□ (30 TIMES)	6	Press any key *except* ENTER 30 times
↵	7	Press ENTER to return from setting the variable to the main layout mode.
↵	8	Press ENTER again to start a new line.

Setting more variables

◆ (HOLD DOWN)	1	Hold down ◆ until the cursor is under the first dot of the second line of dots.
F3	2	Press F3
v	3	Type v
`address2$`	4	Type `address2$`
↵	5	Press ENTER
. (20 TIMES)	6	Type 20 dots
↵ ↵	7	Press ENTER twice

↑ (HOLD DOWN) F3 v address3$ ↵ . (20 TIMES) ↵ ↵

8 Set the variable address3$ under address2$, and make it 20 dots long.

↑ (HOLD DOWN) F3 v address4$ ↵ . (20 times) ↵ ↵

9 Set the variable address4$ under address3$, and make it 20 dots long.

↑ 10 Move the cursor onto one of the dots
Immediately the name of the variable appears in the WORK AREA.

↓ 11 Move the cursor onto the dots of other variables. Each new variable is displayed in the WORK AREA. As soon as the cursor is clear of the dots, the WORK AREA clears.

Note that the only way to set a variable is by using F3 and v. Typing dots on the screen using the full stop key will simply show a line of dots on the screen.

SEDIT and the F3 key

Pressing the F3 key when using sedit offers one of four options, each of which (like edit or the commands in EASEL, QUILL or ABACUS) can be used by pressing a single key followed by ENTER.

V marks the start position for displaying or inputting a variable on the screen. Press any key except ENTER to mark the length of the display. If a variable is already there, using F3 and v deletes it instead of setting another.

When the screen is used, variables set on the screen *do not cause an error* if they do not already exist.

C clears the whole screen layout completely. Be careful using this, as you can't call back your screen if you clear it without having saved it.

I sets the ink colour, which remains in force until you set a new colour, or you leave sedit.

P sets the paper colour, which, like ink, remains in force until you set a new colour, or you leave sedit.

Using cursor keys after setting paper leaves the screen its original paper colour. Using the SPACE BAR 'paints' it in the current paper colour.

Note that in version 1, you cannot use I and P to set the colour of variables, which are always white on black.

Before going any further Miranda decides to save her work.

To leave `sedit`:

ESC 12 Press ESC when in the main layout mode
As soon as you leave `sedit`, the screen clears. The layout you have just created immediately reappears, with the fields of the current record in place of the dots.

Saving screen files – the SSAVE command

`ssave` ↵ 1 Type `ssave` and press ENTER
Note the double 's' at the beginning of the word, which stands for *screen save*.

`wed` ↵ 2 Type `wed` and press ENTER. The right-hand Microdrive whirrs and a copy of the screen layout you have just created is saved onto tape with the file name `wed_scn`.

The command `ssave` allocates the extension `_scn` unless you specify otherwise.

Coping with mistakes in SEDIT

- Most of the normal editing keys do not work in `sedit` in the normal way (if at all).
- The only way to correct anything you type in is by backspacing over it using CTRL + ←, or spacing over it using the SPACE BAR and retyping it.
- The → and ← keys work as expected, but the ↑ and ↓ keys move up and down the screen, rather than to the left and right of the lines.
- While setting a variable, every key except ENTER and backspace produces a dot on the screen.
- You cannot get out of setting a variable by pressing ESC. This merely produces a dot. You must press ENTER to return to layout mode, and then delete the set variable.
- Variables, once set, can be deleted by placing the cursor over any one of its dots and pressing F3 and v and ENTER.
- The only way to change the length of the variable once it is set is to delete it and set it all over again.
- It is possible to move a variable, once set, to a completely new position on the screen without first deleting it. Place the cursor at the start of the new position and set the variable again.
- The position set originally for this variable will be deleted automatically. The dots, however, remain displayed on the `sedit` screen. They can be erased using backspace or SPACE.

USING YOUR OWN SCREEN

```
open ↵guests" logical"g ↵
```

1 Open the `guests` file again.

```
load ↵wedcom ↵
```

2 Load the `wedcom` procedures.

```
comget ↵
```

3 Type `comget` and press **ENTER**
The '`Command ?:`' prompt appears on the screen.

```
n
```

4 Press `n` to change the record
The record is immediately displayed, exactly as if you were using the standard display provided by `display`.

```
s
```

5 Press `s` to stop the procedure.

```
cls ↵
```

6 Type `cls` and press **ENTER**. The screen clears.

```
screen ↵
```

7 Type `screen` and press **ENTER**
Your own screen layout reappears.

```
display ↵
```

8 Type `display` and press **ENTER**
Your own layout is replaced by ARCHIVE's standard display.

The command `display` always shows ARCHIVE's standard screen layout. The command `screen` re-displays whichever screen layout is in temporary memory, whether ARCHIVE's standard layout or one created in `sedit`.

The standard layout has replaced your layout in temporary memory.

To use your layout again or to edit it, it must first be loaded from tape back into memory.

Loading screen files – the SLOAD command

```
sload ↵
```

1 Type `sload` and press **ENTER**. Note the `s` on the front of the word.

```
wed ↵
```

2 Type `wed` and press **ENTER**
The screen clears and the standard layout is replaced by the layout saved in `wed_scn`. The current record is automatically displayed.

```
sedit ↵
```

3 Type `sedit` and press **ENTER**
The current record is replaced by dots representing the set variables.

```
↵ (4 TIMES)
```

4 Press **ENTER** four times and cursor to the end of the `address2$` field.

```
↓ (10 TIMES)
```

5 Press ↓ ten times

```
Coming ? ↓ (6 TIMES)
```

6 Type `Coming ?` followed by six presses of ↓

```
F3
```

7 Press **F3**

`v` 8 Type `v`

`coming$` ↵ 9 Type `coming$` and press **ENTER**

`.` ↵ 10 Type one dot and press **ENTER**

↵ 11 Press **ENTER** again to set the variable on the next line.

↓ (UNTIL UNDER S OF `send`) `No. coming` ↓ ↓ F3 `v partyof` ↵ `..` ↵ ↵

12 Range the next variable underneath this one, lining up the message and setting the variable in the same column (see diagram).

The screen should now look like this:

```
                    WEDDING GUEST LIST

Name      ....................................
Address   ....................................
          ..............................          Coming ?   .
          ........................                No. coming ..
          ........................                           .
```

ESC `ssave` ↵ `wed` ↵

14 Leave `sedit` and `ssave` the new layout to the screen file `wed`.

Note: it is possible to type in more characters in a field (using `alter` or `insert`) than the number of dots entered when the variable is set. The extra letters will remain on the display until cleared by `screen`.

The number of dots does, however, limit the number of characters displayed (as opposed to input). Lines longer than the number of dots are still in the file, but the extra characters are not shown.

This can have unfortunate consequences. For example, using the `c` option of the `comget` file, it is possible to type in `1000`, and only see 10 when the `partyof` field is later displayed by an `sprint`.

ADDING THE NEW SCREEN DISPLAY TO YOUR PROCEDURES

`edit` ↵ TAB (20 TIMES)

1 Use `edit` to alter the procedure `start`.

↓ (5 TIMES) F5 CTRL↓ sload "wed" ↵

2 Change the line `display` so that it reads `sload "wed"`.

The procedure should now look like this:

```
proc start
  cls
  print at 5,10; ink 2; paper 6; " Opening Files .... "
  error closeall
  open "guests"logical "g"
  sload "wed"
  comget
  endproc
```

ESC 3 Leave `edit`.

save ↵ wedcom ↵ 4 `save` the newly changed program file `wedcom`.

Screen files summary

- Screen layouts behave exactly like ARCHIVE's `display` layout, but can be designed to your own format.
- `sedit` can be used to create and edit screen layouts, just as `edit` can with procedures.
- Additional editing commands are called using F3 followed by
  - V to set or delete a variable position
  - C to clear the `sedit` screen
  - I to set the ink colour of prompts, headings and variables
  - P to set their paper colour
- `ssave` saves screen layouts in a `_scn` file on tape.
- `sload` loads a screen layout from tape.
- `sload` replaces any layout created by `display`, and vice versa. Use `screen` to redisplay the layout if it is cleared by `cls`.
- `sinput` allows you to input variables set in the layout. This is demonstrated in the next chapter.

Incompatible versions of screen files

Screen files created and saved using ARCHIVE version 1 cannot be loaded by version 2, as the file structure is different.

Version 2 comes with a SUPERBASIC program called `scrcon_bas` which converts version 1 screen files to the version 2 format. To run this program, type `lrun mdv1_scrcon_bas` from the QL BASIC SCREEN.

SUMMARY

Miranda decides to enhance the task of putting information into the guests file by creating a screen layout which allows her to put her own prompts on the screen, and to place the names of the fields in whichever position on the screen she chooses.

She does this by using the `sedit` command, to create a screen layout. She clears the standard `display` layout from the screen by using the F3 key, followed by `c`.

First of all, she creates a heading for her screen, which she sets to a different colour, using the paper and ink options.

She then sets each variable in turn, after putting a prompt in front of each one, so she knows what information to type where.

She saves her screen file, using `ssave`, otherwise it will only be stored in temporary memory, and will be lost when she leaves ARCHIVE.

Finally, she adds this screen file to the `wedcom` procedures, so it is called by the procedure `start` when the `guests` file is opened.

Helpful hints

- Draw up a screen layout on paper before trying to set prompts and variables on the screen. Mark the line numbers down the side, so that when you are using `at` statements in procedures, you will be able to see at a glance which line each variable is on.
- Write down the field names and consider how long their maximum contents might ever need to be. Remember, a field that is set too short displays only part of a long data item.
- Do not name `_scn` files the same as `_prg` files, or you may one day accidentally wipe one of them out – the command `ssave` is very similar to `save`.
- When setting variables in one colour and prompts in another, set all the prompts first, using the arrow keys to move over spaces where variables are to be set, then set all the variables. This saves having to reset the paint and ink colours for every prompt and every variable.

CHAPTER 11

A New Program from Old Building Blocks

Startling improvements can be gained in a task's ease, speed and accuracy by using ARCHIVE's ability to automatically make complex connections, cross references, checks and calculations.

Many jobs require looking up information in several different lists, tables, catalogues or files. For example, when a customer orders a product, the orders clerk looks up the product price, and the customer address, calculates the sales, debits the customer and reduces the stock of products before filing a copy in the orders file. If there are many products, orders and customers, this process is both tedious and error-prone.

This chapter shows how to:

- Adapt the wedding programs to manage a quite different set of data
- Use two linked files simultaneously
- Automatically calculate fields from other input

- The following ARCHIVE command is introduced:
  `sinput` – waits for input into variables specified in screen layout

Point to remember

- Don't put too many fields in a file – you probably need instead to create a separate, but linked, file.

Note: From now on the instructions and methods used in this book become more complex, and may not be explained in such painstaking detail as in earlier chapters.

Even if you do not wish to implement the techniques now, or do not quite grasp their significance, browse through them anyway.

At a later date, when faced with a new problem you are unsure how to solve, you may remember seeing a possible approach here, and will find these chapters particularly useful.

Example: The toy charts

Harold Blake is production manager at Tees Toys (*'Where dreams come first'*). His latest brainchild, Lawrence Lizard (dubbed 'The Laurosaur' by Nigel) is not selling too well. Bumper sales at Christmas for the newly launched Luny Llama threaten to topple Larry from the conveyor belts into the bin.

Harold plans to amaze the world, and his boss, by showing that the profits on Lizards outperform the heavily discounted Llamas – a clear case of quality versus quantity.

As this involves repetitive and mildly complex calculations, he decides to use ARCHIVE do all the arithmetic.

Most of the work has already been done for him by Miranda. All he has to do is adapt her program.

INSTRUCTIONS

Harold's first task is to set up the data files. He plans to use two files, one to hold the details of the toys, and the other to hold the sales records. This saves repetition of data – there can be lots of sales for one toy.

This 'one-to-many' relationship is common to numerous activities:

- Libraries have many borrowings for each book
- Doctors have many visits from each patient
- Businesses make many sales to each customer
- Trade magazines have many subscribers in each corporation

It is a waste of time to type in the full details of every book, patient, customer and subscriber address every time, so the information is entered once and 'looked up' each time the process is repeated.

It is of course possible that a book is never borrowed, a person never sick, and so on, so there can be no records rather than many.

The fact that there may be many, or there may be none, makes it impractical to create a field for each borrowing, visit etc. There would have to be as many fields as the maximum number of borrowings, visits etc.

The solution is to create two files and link them, looking up data in one according to the data in the other.

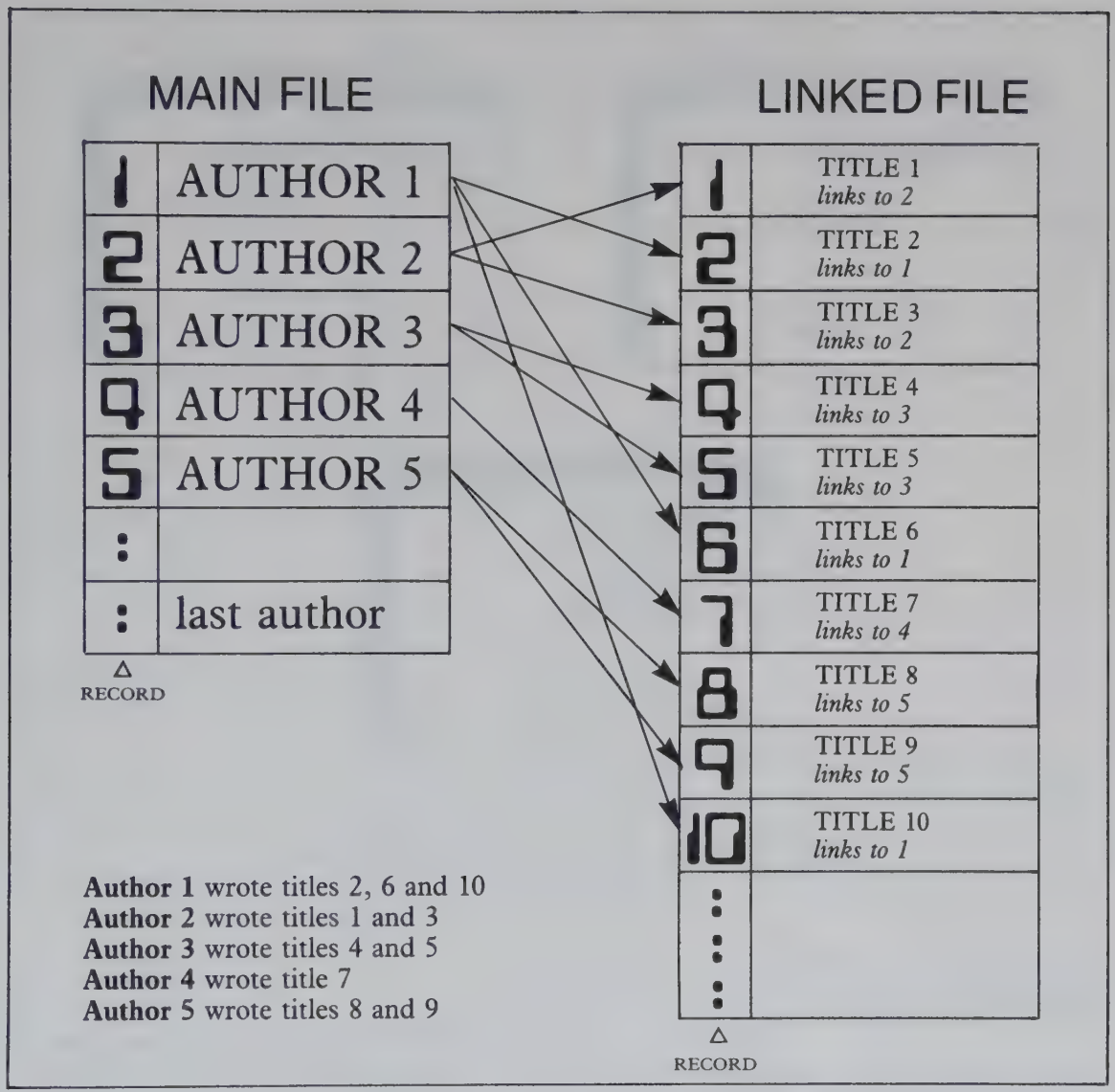


Figure 11.1 - A One-toMany relationship

Multiple one-to-many relationships

This process also allows much more complex relationships to be easily dealt with. In each of the above examples, there can be a second, overlapping one-to-many, or lookup, file:

- Many borrowings for each reader
- Many prescriptions (to patients) for each drug
- Many products in each sale
- Different titles sent to the same subscribers

This chapter only deals with a single lookup and link, but the principle is the same.

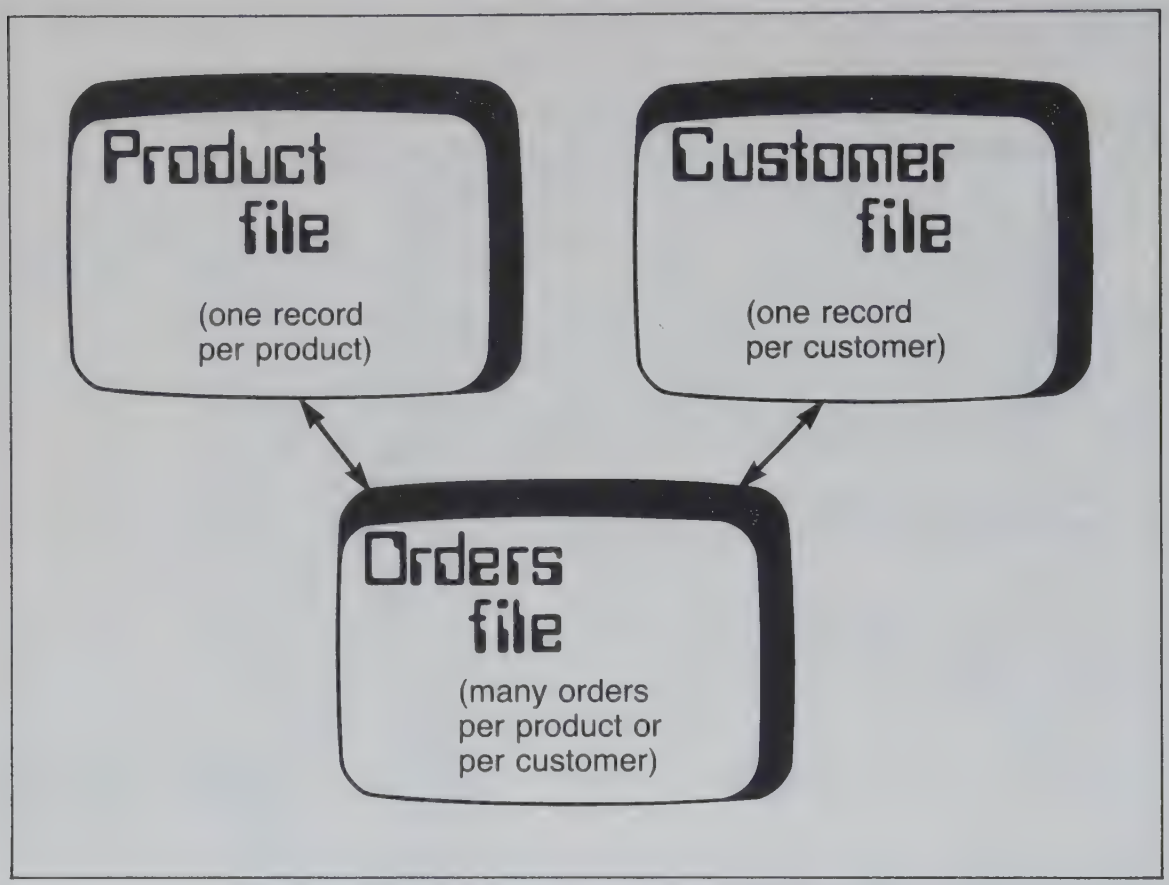


Figure 11.2 – Multiple one-to-many relationships

Starting up

If ARCHIVE is not loaded, load it now. Otherwise make sure all data files and procedures have been cleared from the memory.

Note: from now on, full keystroke commands will not be given, only new lines to type or change will be shown on the left.

The `toys` file

The file `toys` holds the code, name, cost and price of each toy.

```
create "toys" logical "t" ↵
```

1 Start creating the `toys` file.

```
    code$ ↵
```

2 The `code$` field contains a 3 letter code name for each toy. This is used as the 'key' to link with the `sales` file records, which will contain the same 'key'.

```
    name$ ↵
```

3 The `name$` field contains the name of the toy, eg. `Pudgey Penguin`

`cost` ↵ 4 The `cost` field holds the cost of manufacture in pounds as a number. It is used for calculations.

`price` ↵ 5 The `price` field holds the price to the shops as a number.

↵ 6 Press **ENTER** and the word `endcreate` appears. The file has been created.

`display` ↵ 7 Type in `display` and press **ENTER** to see the finished structure.

It should look like this:

```
Logical name :t
code$        :
name$        :
cost         :0
price        :0
```

The `sales` file

Next create the file to hold the records of the monthly sales figures for each toy.

```
create"sales"logical"s" ↵
```

1 Create the `sales` file.

`code$` ↵ 2 The `code$` field contains the code of a toy in the `toys` file. This allows information to be looked up in the toy file when entering a sale.

`month` ↵ 3 The `month` field holds the number of the month the sales were made in, eg. `4` = April.

`qty` ↵ 4 The `qty` field holds the total quantity sold for the month.

`disc` ↵ 5 The `disc` field holds the percentage discount being offered that month on all sales.

`profit` ↵ 6 The `profit` field holds the total profit for the month. This is calculated automatically by the program.

↵ 7 Press **ENTER** and the word `endcreate` appears. The file has been created.

`display` ↵ 8 Type `display` and press **ENTER** to see the finished structure.

It should look like this:

```
Logical name :s
code$        :
month        :0
qty          :0
disc         :0
profit       :0
```

Harold decides that his programs should allow for toy records without any linked sales records, but not the other way around. After all, it is possible to have a toy which does not sell, or has not sold yet, but not the sale of a toy which does not exist.

Other uses for lookups

Lookups as validations

It can be extremely useful to catch mistakes as they are made by checking keyboard input at the time it is entered.

A simple method of doing this, using an `if` clause with logical expressions, was illustrated in Chapter 7.

If the range of possible correct entries is more complex, they can be placed in an ordered file, and the file checked against the input. If it is found, then the entry is correct.

Lookups as code expanders

Codes are quick to type and save space when stored, but are hard to read and remember, especially if they are not your own codes, or you do not use the system frequently.

Create an ordered lookup file of codes, together with the full text description of what they mean, and use it to find the description when coded records are listed or printed.

The toy sales screen layout

Clear the data files from memory by typing `new` and pressing **ENTER**.

Use `sedit` to create a screen layout to show the contents of both files on the same screen.

The variable `f$` also appears on the screen, although not part of a file. It is used to store the name of the file currently in use.

Follow the layout in the diagram. The dots show where variables should be set using **F3** and `v`, and the names in brackets show the relevant variable names to be set there.

Do not actually type the variable names in brackets onto the display itself. They are shown merely to help you create the layout.

- Note the use of colons at either end of the string fields `code$` and `name$`. When the layout is in use, these will show whoever is using the computer how long the fields should be at most.

The ninth line (number 8, since ARCHIVE counts them from 0) holds all the numeric variables for the `sales` file. They are on one line so that they can be easily cleared.

These blank lines at the bottom of the screen are reserved for the command line, prompts and warning messages.

```
.............................................................
                          TOY SALES
                          ---------
Toy Code  :...:    (code$)                File  ..... (f$)
Name      :.....................: (name$)
Cost    £  .....   (cost)
Price   £  .....   (price)

Month      Quantity Sold      Discount %     £ PROFIT
 ..            ...               .....        ........
(month)       (qty)              (disc)        (profit)

.............................................................
```

`ssave` the layout to a file called `toysales`.

- Do not give the same name to `scn` files as `_prg` files. One day you may accidentally leave an 's' off `ssave` and wipe out a set of procedures on tape with any that happen to be in memory at the time. Even if there are no procedures in memory, `save` saves an empty `_prg` file.

When finished the screen should look like this:

```
.............................................................
                          TOY SALES
                          ---------
Toy Code  :   :                              File
Name      :                  :
Cost    £
Price   £

Month      Quantity Sold      Discount %     £ PROFIT









.............................................................
```

CONVERTING 'WEDCOM' FOR TOY SALES

Harold makes a straight copy of Miranda's programs by loading them, making one or two cosmetic alterations, and then resaving them with a new name.

`load` ↵ `wedcom` ↵ 1 Load Miranda's wedding procedures.

`edit` ↵ TAB ↓ F5 SHIFT ↓ CTRL ↓ `TOYS_PRG 12 May 84` ↵
2 Change the `rem` statement in the procedure `a1` to suit the new program, eg:

```
rem TOYS_PRG 12 May 84
```

Delete the lines referring to `coming` and `goto` in `a1`.

TAB (REPEAT TO `coming`) F3 `d` ↵
3 Delete the procedure `coming`, as it does not apply.

TAB (REPEAT TO `goto`) F3 `d` ↵
4 Delete the procedure `goto`, which was created in last chapter's exercises. It is not really very useful, as record numbers can change.

5 Delete the relevant lines from the `comcheck` and the `help` procedures.

ESC 6 Leave `edit`.

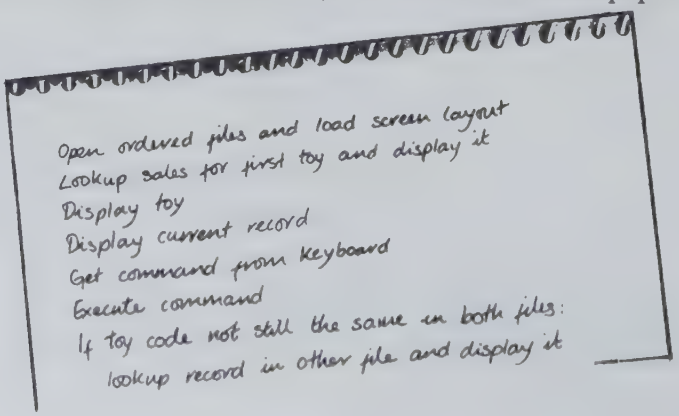
`save "toys"` ↵ 7 Type `save "toys"` and press **ENTER** to save the procedures in a new file under a new name. This should contain the following procedures: `a`, `a1`, `b`, `blank`, `closeall`, `comcheck`, `comget`, `dele`, `errchk`, `escmsg`, `f`, `help`, `i`, `jump`, `n`, `prompt`, `reverse`, `seemore`, `start`, `warning`, `zoom`.

Examining the task

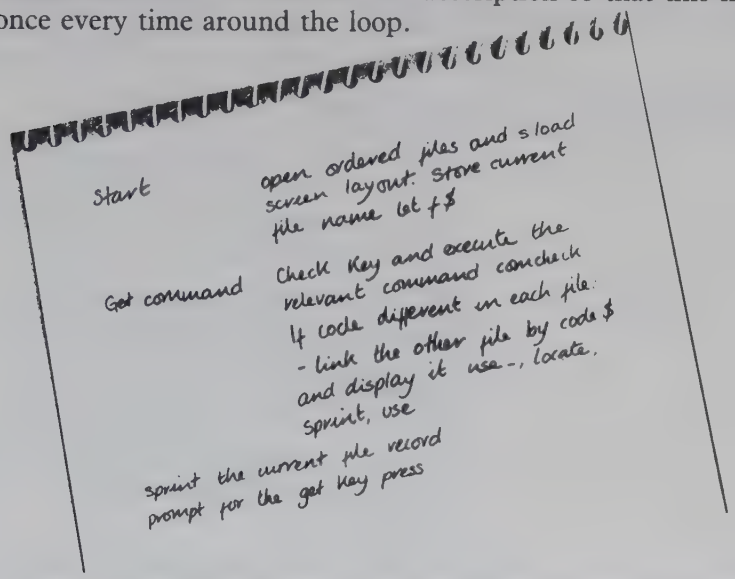
Harold is pleased with his progress so far. All those procedures tucked away in his very own program file, and it only took him a couple of minutes. He fixes himself a large G & T to celebrate, and to fortify him for the task ahead.

Getting started

First, Harold describes the job in broad terms on paper.



Looking at the description, he sees that the starting and finishing look-ups are both simply links between the current file and the non-current file. He rewrites the description so that this happens once every time around the loop.



So far, the only significant departure from Miranda's procedures (`start` and `comget`) is the need to keep two ordered files linked by the same key.

Harold starts in earnest at the beginning, with the `start` procedure.

`edit` ↵ `TAB`

1 Use `edit` and press the tab key to get down to the procedure `start`.

2 Adapt `start` so that instead of opening the `guests` file as `g`, it opens the `sales` file as `s`.

`F4` `order code$;a,month;a` ↵

3 Insert a line to order the `sales` file on both the `code$` and `month` fields, in ascending order. Type in `order code$;a,month;a` and press **ENTER**. Both files need to be ordered on code$ because the toy code is used as a key to link sales to an individual toy. Also, `locate` (used with ordered files) is faster than `find` or `search`.

Sales can be entered into the file in any order for any toy, but sales for the same toy need to appear in sequence when linked to the toy record.

`open "toys" logical "t"` ↵ `order code$;a` ↵

4 Insert another two lines to open the `toys` file as `t`, and to order it on `code$` in ascending order.

Note: The two `order` lines are included here for clarity, and are actually redundant after the files have been ordered once. The order of a file is saved when it is closed (if opened with `open`), and loaded in the same order when the file is re-opened.

Harold now sets up the first values of two variables which will be used to keep track of which record in which file is the current record at any time during the program. These will be changed in the ensuing procedures as the files are swapped, but must be declared here before they can be referred to elsewhere.

`let f$="TOYS"`

5 Insert the line `let f$="TOYS"`

The variable `f$` is used as a message on the screen to remind you which file you are currently using. Most commands will affect the current file, and not the other one, so you need to know which is in use.

The variable is also used as a *flag* by the procedures, to determine which is the current file.

6 Change the line which loads the screen file, so that it loads the screen file `toysales` instead of `wed`.

The screen layout is loaded last of all because its appearance on the screen will signal that the program is ready to receive commands.

The finished procedure should look like this:

```
proc start
  cls
  print at 5,10; ink 2; paper 6; " Opening Files .... ";
  error closeall
  open "sales"logical "s"
  order code$; a,month; a
  open "toys"logical "t"
  order code$; a
  let f$="TOYS"
  sload "toysales"
  comget
  endproc
```

Changing the current file

Harold creates a procedure to switch the current file. Otherwise the `toys` file will always be the current file, and none of the available command options will affect the `sales` file.

He needs a procedure which will switch use to whichever file is not the current file. He uses the variable `f$` to find out which is the current file, and then changes it to the other file to keep it accurate.

`F3` `n xchange` ↵ `if f$="SALES"` ↵

1 Create a procedure called `xchange`. The first line should read `if f$="SALES"`, ie. if the current file is the `sales` file.

`use "t"` ↵

2 The second line reads `use "t"`
If `sales` is the current file, then use the `toys` file.

`let f$="TOYS"`

3 Add a line to reset the variable `f$` to match the current file, `toys`.

`else` ↵

4 The next line is `else`, (meaning 'otherwise'): ie. if the current file is not `sales`, do the following commands.

`use "s"` ↵

5 Add a line which sets the current file to `sales`.

`let f$="SALES"` ↵

6 Add the line to get `f$`: `let f$="SALES"`

`endif` ↵ ↵

7 End the `if` clause: `endif` and leave insert mode.

The finished procedure should look like this:

```
proc xchange
  if f$="SALES"
    use "t"
    let f$="TOYS"
    else
    use "s"
    let f$="SALES"
    endif
  endproc
```

The procedure which checks command letters, and the `help` and `a1` (remarks) procedures, all need to be altered to take account of the new procedure.

8 Insert `if c$="X":xchange:endif` into `comcheck`

The procedure should now look like this:

```
proc comcheck
  if c$="X": xchange: endif
  if c$="?": help: endif
  if c$="S": stop : endif
  if c$="F": f: endif
  if c$="N": n: endif
  if c$="B": b: endif
  if c$="A": error a: errchk: endif
  if c$="I": error i: errchk: endif
  if c$="D": dele: endif
  if c$="1": first: endif
  if c$="Z": zoom: endif
  if c$="R": reverse: endif
  if c$="J": jump: endif
  endproc
```

9 Don't forget to update the `help` and `a1` procedure. Add an option for `X`:

```
print "X change the current file" to help
```

10 Add this to a1:

```
REM xchange: (comcheck), uses other file, resets
f$
```

11 Update the rem line in the procedure a1 to the name TOYS1_PRG and save the procedures to toys1.

Linking the files

The program can now be employed to use all of Miranda's procedures on either file independently, but the information displayed will rarely be for the same toy unless the records are forced to remain linked.

This is obviously misleading, since both records on the screen need to be for the same toy, especially as the primary purpose of having two files is to look up toy information for every sale.

An additional three lines need to be added to the procedure comget to reflect the fact that two files are now in use.

After every command, ARCHIVE returns to comget. If, as the result of some command, the sales and toy records no longer both relate to the same toy, this is corrected here. The link is checked, and a new link made if necessary.

1 Edit the comget procedure.

2 Insert let othercode$="" before the while 1 line:

The variable othercode$ is used to store the current code$ in the file not currently in use. It is used to detect when the code$ field in the current file (the value of code$ itself) is not the same as the code$ in the other file (stored in othercode$).

Here othercode$ is only set to nothing the first time that comget is used, because the files are not yet linked.

3 Come down one line and insert the following three lines:

```
if code$<>othercode$
  linkrecs
  endif
```

This if clause calls the procedure linkrecs if the current code$ field is not the same as the code held in the variable othercode$ (representing code$ in the non-current file).

The first time comget is used it will link a sales record with the current toy record, since othercode$ starts as "", which cannot be a code$ in either file.

Later, if an action taken from comcheck moves one file out of

step, the relevant record in the other file is located, to keep sales and toy records linked by the same toy code.

The procedure `comget` should now look like this:

```
proc comget
  let othercode$=""
  while 1
    if code$<>othercode$
      linkrecs
      endif
    sprint
    blank; 13,5
    print at 12,5; "Command ?: "; paper 6; " "; paper 0; " ? for help"
    let c$=upper(getkey())
    print at 12,16; c$
    blank; 14,5
    comcheck
    endwhile
  endproc
```

The program should now contain the following procedures:

```
a           errchk      reverse
a1          escmsg      seemore
b           f           start
blank       help        warning
closeall    i           xchange
comcheck    jump        zoom
comget      n
dele        prompt
```

The LINKRECS procedure

Deciding which file to link

Harold realizes that whichever file is the current file, the other will have to be linked to it. If they are not linked with the same toy code, this is detected in `comget`. The procedure `linkrecs` performs the link. He makes use of the variable `f$` (which should always contain the current file name) to determine whether he has to link sales to toys, or the other way round.

F3 `n linkrecs` ↵ `let othercode$=code$` ↵

1 Create a procedure called `linkrecs`, and type the first line `let othercode$=code$`.

Note that the value of `othercode$` is set to the code in the current file before the files are swapped. This assumes that a similar code in the other file will be found. If it isn't, `othercode$` is changed.

`if f$="SALES" ↵` 2 The second line should be typed in as follows:

```
if f$="SALES"
```

The variable `f$` is used to determine which file is the current file. The other file is then used and linked.

`linktoys; code$ ↵` 3 Line three is `linktoys; code$`

The value of the current `code$` field is passed as a parameter to a procedure linking toys to sales using the code field.

`else ↵ linksale; code$ ↵`

4 Type the following lines to link sales to toys if the sales file is not the current file:

```
else
linksale; code$
```

`endif ↵ ↵` 5 Close the `if` clause: `endif`, and leave insert by pressing **ENTER**

The procedure now looks like this:

```
proc linkrecs
  let othercode$=code$
  if f$="SALES"
    linktoys; code$
    else
    linksale; code$
    endif
  endproc
```

Linking `toys` with `sales`

The procedure `linkrecs` requires two possible options – linking toys to sales, and sales to toys. They are separate so that they can be used by other procedures later, and because (by Harold's reckoning) there might (at first) be no sales for a toy. The procedure which links sales to toys must take account of this possibility.

Harold starts with the easy one and creates the procedure to link a toy record to a sales record: he needs to locate the same code in the `toys` file as is in the current `sales` record. First he has to make `toys` the current file.

`proc linktoys; seek$ ↵`

1 The procedure to do this is called `linktoys; seek$`. The value of `seek$` will be the same as `code$` when the procedure is called with `code$` as a parameter.

`use "t"` ↵ 2 The second line makes `toys` the current file. The current file is `sales`, so `toys` must be made the current file to find a linked `toys` record.

`locate seek$` ↵ 3 The next line reads: `locate seek$`
The command `locate` can be used when a file has been ordered. It jumps immediately to the right position in the file, and is therefore much faster than `find` or `search`.

The file is supposed to have a toy for every sale, so the `locate` should never fail.

`sprint` ↵ 4 To display the current field when the record has been found, `sprint` is used.

`sprint` only affects the current file, so the sales values are not redisplayed.

`use "s"` ↵ 5 Before linking files, the `sales` file was the current file. It is made current again after the link has been made and displayed.

The procedure should look like this:

```
proc linktoys; seek$
  use "t"
  locate seek$
  sprint
  use "s"
  endproc
```

Linking `sales` with `toys`

The procedure for linking a `sales` record to a toy record works in exactly the same way, but is significantly different. It is possible for there to be no sales records for a given toy.

If there are no `sales` records with the current toy code, `locate` will find the next sales record in order, which will not have the same code as the toy record.

The command `sprint` used with the `sales` file would then display the wrong record, one which does not relate to the displayed toy. This could have unfortunate consequences.

Harold gets round this problem by blanking the sales line instead of using `sprint`.

Harold sets `othercode$` to nothing if there are no sales records.

`linksale; seek$` ↵ `use "s"` ↵ `locate seek$` ↵ `if s.code$<>seek$` ↵
1 Create a procedure called `linksale; seek$` and type the following lines:

```
linksale; seek$
use "s"
locate seek$
if s.code$<>seek$
```

If there is no linked record in the `sales` file, `locate` will leave a record with a different `code$` as the current record.

It is not wanted here.

In ARCHIVE version 2, you can test if `locate` has been successful by using `found()` with this `if` clause.

```
blank; 8,0
```

2 Add the following line: `blank; 8,0`
This line blanks out the previous toy values displayed in the layout. A `sprint` now would show a sale unrelated to the toy on display.

```
let othercode$=""
```

3 The next line is `let othercode$=""`
The variable `othercode$` is set to a null string, as there is no current toy code in the sales file. This avoids problems with linking in `comget`.

```
else ↵sprint ↵endif ↵use "t" ↵
```

4 Add the following lines to the procedure:

```
else
sprint
endif
use "t"
```

↵ 5 Leave `insert` mode.

The finished procedure should look like this:

```
proc linksale; seek$
  use "s"
  locate seek$
  if s.code$<>seek$
    blank; 8,0
    let othercode$=""
    else
    sprint
    endif
  use "t"
  endproc
```

6 Update the procedure `a1` so that the `rem` line reads `TOYS2_PRG`. Add information about `linkrecs`, `linksales` and `linktoys`,

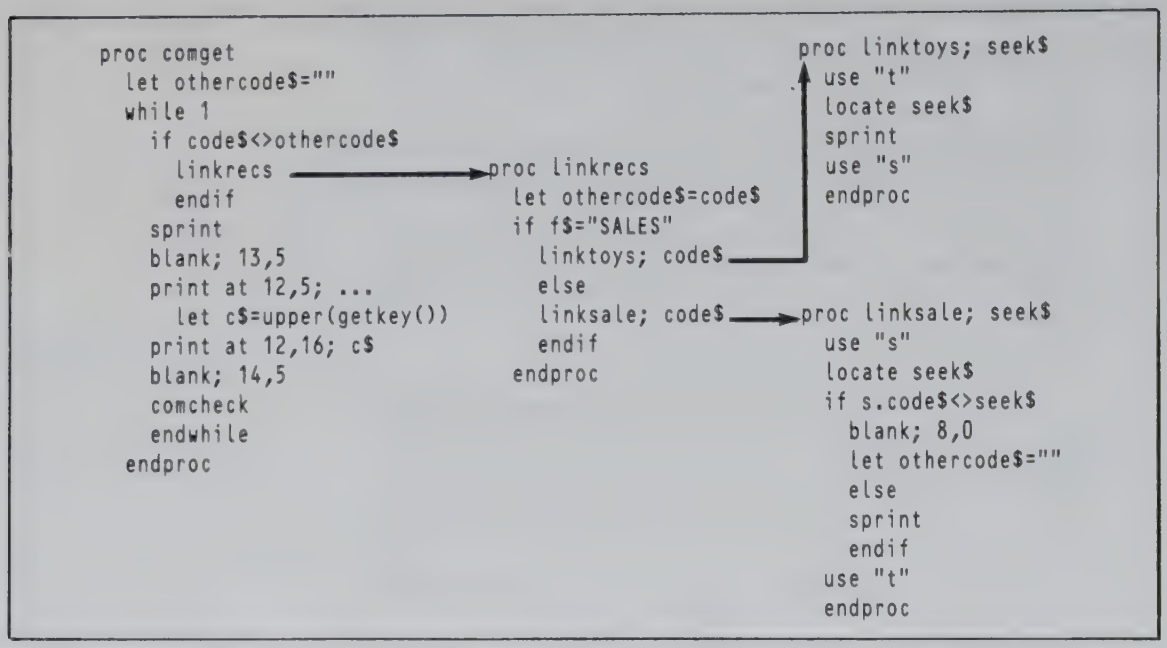
The procedure should now look something like this:

```
proc a1
  rem      TOYS2_PRG  12 May 1984
  REM a: (comcheck), (error) alter
  REM b: (comcheck), back, warning if start of file
  REM blank: (comget) (linksales), blanks rest of line at co-ordinate x,y
  REM closeall: (comget), closes all open files
  REM comget: controlling loop menu, sprints current, getkey command c$
  REM comcheck: (comget), uses if to test c$
  REM dele: (comcheck), deletes after confirm (if no sales when toy)
  REM errchk: errnum() warning
  REM escmsg: prints escape message
  REM f: (comcheck), input and find txt$, calls seemore
  REM help: (comcheck), lists available options, cls and screen
  REM i: (comcheck as error), insert
  REM jump: (comcheck), input num, position recnum()+num
  REM linkrecs: (comget), locate code$ in other file
  REM linksales: (linkrecs), locates t.code$ in sales or blanks
  REM if othercode$=""
  REM linktoys: (linkrecs) (saleins), locates s.code$ in toys
  REM n: (comcheck), next, warning end of file
  REM prompt: print a prompt (msg$) at 13,5
  REM reverse: (comcheck), back while recnum()>0
  REM seemore: (f): continue loop while found()
  REM start: closes all files, opens, sload files, sets f$
  REM warning: print warning (msg$) at 14,5
  REM xchange: (comcheck), uses other file, resets f$
  REM zoom: (comcheck), next loop while not eof
  endproc
```

7 Save the procedures to `toys2`.

Notice the way that all the procedures associated with linking begin with the same letters. This way, following the course of a link, or trying to trace mistakes, does not necessitate tabbing around all the procedures in the file.

The illustration at the top of the next page shows how the files are linked together:



Some records to get started

Harold can hardly wait to try out his obviously inspired programming, despite the fact that the program as a whole is not yet finished.

He uses `start` to get the ball rolling, then presses `i` while using the `toys` file to insert some toys.

He is very careful to make sure that the toy codes (`code$`) are consistently in capital letters, so that they will always be the same for `locate`.

He types in details for the following toys:

```
LLU  Luny Llama          3.45      8.5
LLZ  Lawrence Lizard     2.34      9.99
PUD  Pudgey Penguin      1.78      5.5
```

The data should be stored in the fields in this format:

```
code$          :PUD
name$          :Pudgey Penguin
cost           :1.78
price          :5.5
```

Harold does not add any sales records at this stage, because he will have to work out the profit (the main point of this exercise) by hand.

He's not too pleased with the way toys are inserted into the file either. `insert` does not force the code into upper case, making

him use the SHIFT key, and there is nothing to stop him from using the same toy code twice, which could be disastrous for the accuracy of the figures.

The SINPUT command

Harold needs some way of putting data in fields one at a time, rather than altogether, as with `amend` and `insert`. He can check each field in turn before proceeding with further input (if any). He decides to use the command `sinput`.

The command `sinput` is very similar to the command `input`. It causes ARCHIVE to wait for you to type in (input) data in the DISPLAY AREA, which is then stored in a variable.

`sinput` only works with variables which are named in the currently displayed screen layout, whether it is one of your own `_scn` files, or `display,s` standard layout.

`sinput` does not need prompts or print items (such as `at` and `ink`), as this is taken care of by the screen layout.

Unlike `input`, the command `sinput` does not clear the current value of the variable before fetching your input from the screen.

It is used by following the command with a list of the fields or variables that you want to enter, like this:

```
sinput name$,cost,price
```

Inserting a new toy

Harold writes a set of procedures to check new toy codes before a record is inserted into the file. After converting the code to upper case and checking that it is not already in use, the procedures then fetch the remaining fields from the screen one at a time, using `sinput`. The record is then appended.

This prevents the same toy code from belonging to different toys.

He plans the four procedures as follows:

`toyins`	– called from `comcheck`, inserts a toy record if the code$ is new. It calls
`gettoycode`	– prompts for the toy code and converts it to upper case
`toyclear`	– clears the field variables for input
`toyinput`	– uses `sinput` to let you put data in the fields

`proc toyins ⏎gettoycode ⏎`

1 Create the first of these procedures, `toyins`. The first two lines are:

```
proc toyins
  gettoycode
```

The procedure `gettoycode` prompts for the toy code to be input, and converts it to upper case.

```
locate newcode$
```

2 Type line 3: `locate newcode$`
The command `locate` is used to find the code if it already exists.

```
if code$=newcode$
```

3 The next line is `if code$=newcode$`
This line checks if `locate` has found a record. If the current `code$` is not the same as `newcode$`, which was the wanted code, then it has not found it.

In ARCHIVE version 2, the same check can be achieved with the function `found()`, which does not work with `locate` in version 1.

```
warning;"Code already in use"
```

4 Now a line should be added to warn the operator that the toy code just entered already exists:

```
warning; "Code already in use"
```

- Each toy code must be unique, or the figures will come out wrong, with sales linked to the wrong toys. If the code exists, don't insert a new record with the same code.

```
else ↵let code$=newcode$ ↵
```

5 The next two lines are as follows:

```
else
let code$=newcode$
```

This line makes the current value of `code$` the same as `newcode$`, once it has been checked.

```
toyclear
```

6 Now call the procedure `toyclear`
This procedure sets the field variables to zero, and `sprints` them.

```
error toyinput
```

7 The next line calls the procedure `toyinput`, which fetches input from the screen for the remaining fields.

It is called with the command `error` so that the input can be abandoned using ESC, and errors (such as putting letters in a numeric field) do not collapse the whole program.

```
errchk
```

8 Next add a line to check for errors:

```
errchk
```

Always check for errors after a procedure has been called with `error`. The `errchk` procedure will report any errors.

If some fields have already been input when a new record is abandoned, the record will not be appended. The error message will indicate why.

`endif` 9 Finish off the procedure by closing the `if` clause: `endif`

The finished procedure should now look like this:

```
proc toyins
  gettoycode
  locate newcode$
  if code$=newcode$
    warning; "Code already in use"
    else
    let code$=newcode$
    toyclear
    error toyinput
    errchk
    endif
  endproc
```

10 Press ENTER to leave insert mode in order to create the next procedure.

Prompt for a new code

The next procedure, `gettoycode`, is very straightforward, and prompts for further input on line 13, as elsewhere in the program, using the procedure `prompt`.

It is essential that the toy codes are always stored consistently, so that the correct links beween the files can be made.

The procedure looks like this:

```
proc gettoycode
  prompt; "Toy code ? "
  input newcode$
  let newcode$=upper(newcode$)
  endproc
```

Clear the fields for input

1 Now create a procedure, `toyclear` to clear and display the values of the current fields.

```
toyclear
let name$=""
let cost=0
let price=0
sprint
endproc
```

The screen layout shows the values of the last record displayed unless the command `sprint` is used. If the field values are not

cleared and shown before input, it appears you are overwriting another record.

Note that setting the field variables to 0 does not actually affect the file until the commands `append` or `update` are used.

Input values for the `toys` file

```
sinput name$,cost,price
```

1 Finally create the procedure `toyinput` to get the values for the `toys` file. The first line is `sinput name$,cost,price`.

The field `code$` has already been allocated the correct value in `toyins`. The other fields are fetched here.

```
append
```

2 The second line adds the input variables to the `toys` file: `append`.

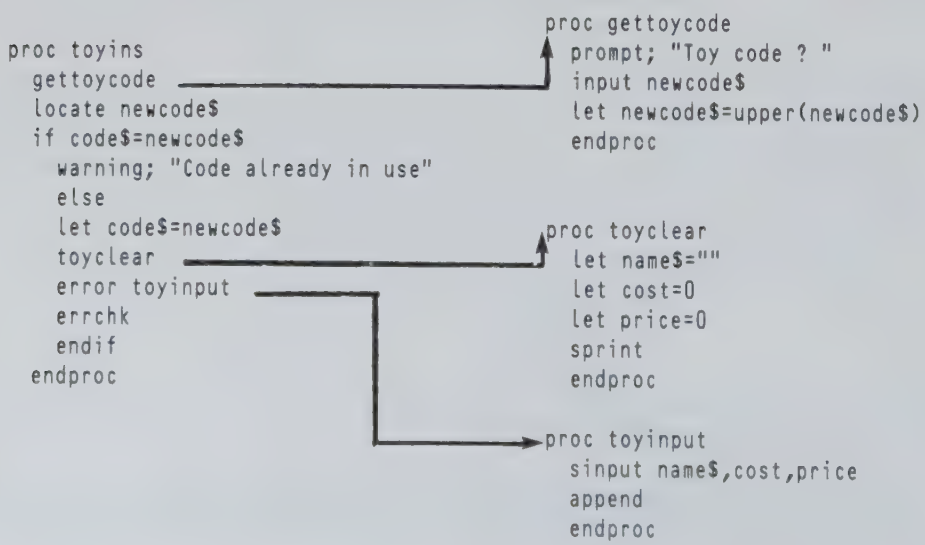
If `sinput` is abandoned because of an error, or an **ESCAPE**, the procedure collapses before it ever reaches this line, and the record is not appended.

However, some of the field variables shown will be wrong until the next `sprint` is performed.

The finished procedure now looks like this:

```
proc toyinput
  sinput name$,cost,price
  append
  endproc
```

The procedures work together like this:



Tidying up

1 Now that a procedure has been written which adds records to the `toys` file, the procedure `i` is no longer necessary, so delete it.

2 Change the procedure `comcheck` by cutting this line:
```
if c$="I":error i:errchk:endif
```

3 Insert 3 lines instead:

```
if c$="I"
if f$="SALES":saleins:else:toyins:endif
endif
```

Now that `i` has been deleted, a new procedure for inserting sales records will have to be written.

If the variable `f$` indicates that `sales` is the current file, `saleins` inserts a sale. Otherwise, `toyins` is used.

4 Update the `a1` procedure to include `toyins`, `gettoycode`, `toyclear` and `toyinput`:

```
REM gettoycode: (toyins) (saleins), prompts for upper(toycode)
REM toyclear: (toyins), clears field variables
REM toyinput: (toyins) (comcheck), sinput fields except code$
REM toyins: (comcheck), append if code$ is new
```

Inserting and calculating a new sale

Harold now comes to the *piece de resistance* of his handiwork: the automatic calculation of profit whenever he enters a monthly sales figure.

He uses `sinput` to control the way sales are entered, in much the same way as for toys.

The initial procedure for inserting a new sale, `saleins`, is very similar to `toyins`. The key difference is that this time the code must already exist in the toy file. If it doesn't, there is nowhere the program can look up the relevant cost and price for the calculation.

`gettoycode ↵ linktoys; newcode$ ↵`

1 The first two lines of the procedure `saleins` are as follows:

```
gettoycode
linktoys; newcode$
```

The procedure `linktoys` performs the necessary `locate` in the `toys` file.

It was originally created for use with `linkrecs`, but serves the same purpose here.

```
if t.code$=newcode$
```

2 The third line is: `if t.code$=newcode$`
Note the use of the logical file name `t` with the variable `code$`. This is needed because the current file is `s`, which also has a field called `code$`.

If the current toy `code$` is not the one looked for, the procedure displays a warning and does not continue with inserting a sale.

```
let code$=newcode$ ↵saleclear ↵error saleinput ↵
```

3 The next three lines are as follows:

```
let code$=newcode$
saleclear
error saleinput
```

The procedure `saleinput` uses `sinput` to fetch the month, discount and quantity of toys sold, then calculates the profit and appends the record.

```
errchk ↵else ↵warning;"No such toy code" ↵endif ↵
```

4 Now finish off the procedure by adding an `else` clause to print a warning message, and end the `if` clause.

The finished procedure looks like this:

```
proc saleins
  gettoycode
  linktoys; newcode$
  if t.code$=newcode$
    let code$=newcode$
    saleclear
    error saleinput
    errchk
    else
    warning; "No such toy code"
    endif
  endproc
```

Type in `saleclear`, the procedure for clearing the field values and redisplaying the zero values.

The procedure should look like this:

```
proc saleclear
  let month=0
  let qty=0
  let disc=0
  let profit=0
  sprint
  endproc
```

The calculation

The procedure `saleinput`, like `toyinput`, uses `sinput` to let you type new values for the fields on the screen. It then uses these figures together with data from the current toy record to calculate the profit on the sales.

Before starting, Harold analyses the steps in the calculation. The profit from any one sale will be the price of the toy, less any discount, less the cost of making it:

```
Price - Discount - Cost = Profit
```

The discount must be calculated first, since it is typed in as a percentage, eg. `20%`. A sum less `20%` is the same as `80%` (100–20) of that sum, so Harold adjusts the formula to look like this:

```
Price x (1 - Discount% / 100) - Cost = Profit
```

If the discount percentage is `20%`, this results in the price being multiplied by `.80`, which is the same as subtracting `.20` times the price.

Finally, this profit per toy has to be multiplied by the quantity sold to give the total profit on the sale.

```
sinput month,qty,disc
```

1 Create the procedure `saleinput` and type the first line as follows:

```
sinput month,qty,disc
```

Note that the `profit` field, like the `code$` field, is not in this list, and so is never changed directly. It is calculated in the next line.

```
let profit=qty*(t.price*(1-disc/100)-t.cost)
```

2 Type the next line which performs the calculation:

```
let profit=qty*(t.price*(1-disc/100)-t.cost)
```

The two fields from the `toys` file which are used in the calculation are identified by the logical filename `t`.

- Note that ARCHIVE works out arithmetic calculations step by step according to certain priorities (just as it does logical expressions):
- Expressions in brackets come first, multiplication and division come before plus and minus
- Without the brackets, the priorities of the other operators would cause this calculation to come out wrong
- If in doubt, bracket the separate parts of complex expressions. It makes them easier to read as well

`append ↵ sprint ↵` 3 Finish off the procedure with the following two lines:

```
append
sprint
```

The finished procedure looks like this:

```
proc saleinput
  sinput month,qty,disc
  let profit=qty*(t.price*(1-disc/100)-t.cost)
  append
  sprint
  endproc
```

4 Update the `rem` line in the a procedure so it reads `TOYS3_PRG`, and add `rem` lines for each of the procedures just added:

```
REM saleclear: (saleins), clears field variables
REM saleinput: (saleins), uses sinput to put data in fields, calcs profit
REM saleins: (comcheck), linktoys
```

5 Leave `edit` and save the procedures to the program file `toys3`.

Operation priorities

ARCHIVE calculates complex expressions step by step, but not in an arbitrary, simply sequential sequence. Each part of an expression has a priority rating:

Operation	Priority
Subscripting and slicing	12
All functions	11
^ (exponentiation)	10
Unary minus	9
()	8
*, /	7
+, -	6
=, <, >, <=,>=, <>	5
not	4
and	3
or	2

Operations with a high number are performed before lower priority ones.

Note that brackets effectively override all mathematical, relative and logical operators. Use them to ensure accuracy and to make expressions easier to read.

Sales of Larry Lizard

Harold now sets about entering the figures for the sales of Lawrence Lizard and Luny Llama. There are only six months' figures for Luny, who was launched upon an unsuspecting world in July.

In fact, Harold blames irresponsible Llama sales promotions by the marketing department for Larry's disappointing figures in July. Harold hopes his comparison of their average profits will restore Larry, and Harold himself, to grace.

1 Type in the command `comget` to get the program going again.

   There is no need to `start` again, waiting while all the files are closed and reopened, as they are already open and all the variables needed by the procedures are set.

2 Press `x` to change from the `toys` to the `sales` file.

3 Press `i` to insert a sale.

`llz` 4 Type in `llz` and press **ENTER**

5 Enter the following figures:

```
Month    Quantity     Discount %

Lawrence Lizard - Code LLZ

Jan         10
Feb        180
Mar        310
Apr        320
May        230
Jun        260
Jul        -20 (including returned lizards)
Aug        210            20
Sep        180
Oct        135
Nov        120
Dec         90

Luny Llama - Code LLU

Jul        525            50
Aug        475            50
Sep        330            20
Oct        260            20
Nov        210
Dec        300            50
```

Harold now has the figures and monthly profits for the two toys. In the next chapter he adapts Miranda's listing procedures to total the figures on either the printer or the screen.

1 Change the `rem` line in the procedure `a` so that it reads `TOYS_PRG`

2 Save the procedures as `toys`

3 `kill` the program files `toys1`, `toys2` and `toys3`

`dir` ↵ ↵ 4 Do a directory of drive 2 to check the date on the `_lab` file.

`dir` ↵ `mdv1` ↵ 5 Do a directory of your backup tape, in drive 1, to check the date on the `_lab` file to make sure it is the correct backup tape. If it is the same as your working tape, use your other backup tape.

`new` ↵ 6 Clear all procedures and data files from temporary memory by typing `new` and pressing **ENTER**

`backup ...` 7 Backup files created or changed since the tape was last used.

`save "May12_lab"` ↵ F5 SHIFT ◗ `mdv1_` ↵ 8 Save label files onto both the working tape and the backup tape in drive 1

9 and `kill` the old `_lab` files on both tapes.

SUMMARY

Harold thinks it would be a good idea to adapt some of Miranda's procedures for his own ends. He wants to compare sales of Lawrence Lizard and Luny Llama. He creates two new files, `toys` and `sales` which both contain a common field, `code$`, so they can be linked together. He also designs his own screen layout, using `sedit`.

Now he sets about adapting Miranda's `wedcom` procedures. He adapts the `start` procedure so that it opens his two new files, ordered on `code$`, (and month in the case of the `sales` file). He then creates a procedure to enable him to change from one file to the other, by pressing one letter. He calls this procedure `xchange`.

Now Harold needs to link the two files together, so that the correct sales figures appear for the each toy. He writes three procedures to do this, `linkrecs`, `linktoys` and `linksale`. He saves his procedures and tries them out by entering some sales figures for Luny and Lawrence.

He then uses the command `sinput` to update the `toys` file when all the details have been entered on the screen.

Finally, Harold writes an input procedure for the `sales` file which automatically calculates the profit from the price, the cost and the amount of discount.

Helpful hints

- Use more files, and fewer fields per file. Link the files.
- Use variables as 'flags' to keep you (and your procedures) informed about important conditions, such as the current file name, or the linked record in other files.
- Use brackets for clarity and accuracy in complex expressions.
- Use `sinput` (even with `display`'s standard layout) to control input field by field.
- Catch data mistakes as they are made during input. They are much harder to find and correct later, and may not be noticed until after they have caused much trouble.
- Use ordered 'lookup' files for linking records, validating input and expanding codes.
- Put colons or similar delimiters either side of field variables set in `sedit` to show how long the expected maximum input should be.

Program listing

`toys_prg` should now contains the following procedure:

```
proc a
  errchk
  alter
  endproc
proc a1
  rem      TOYS2_PRG  12 May 1984
  REM a: (comcheck), (error) alter
  REM b: (comcheck), back, warning if start of file
  REM blank: (comget) (linksales), blanks rest of line at co-ordinate x,y
  REM closeall: (comget), closes all open files
  REM comget: controlling loop menu, sprints current, getkey command c$
  REM comcheck: (comget), uses if to test c$
  REM dele: (comcheck), deletes after confirm (if no sales when toy)
  REM errchk: errnum() warning
  REM escmsg: prints escape message
  REM f: (comcheck), input and find txt$, calls seemore
  REM gettoycode: (toyins) (saleins), prompts for upper(toycode)
  REM help: (comcheck), lists available options, cls and screen
  REM i: (comcheck as error), insert
  REM jump: (comcheck), input num, position recnum()+num
  REM linkrecs: (comget), locate code$ in other file
  REM linksales: (linkrecs), locates t.code$ in sales or blanks
  REM if othercode$=""
  REM linktoys: (linkrecs) (saleins), locates s.code$ in toys
  REM n: (comcheck), next, warning end of file
  REM prompt: print a prompt (msg$) at 13,5
```

```
  REM reverse: (comcheck), back while recnum()>0
  REM saleclear: (saleins), clears field variables
  REM saleinput: (saleins), uses sinput to put data in fields, calcs profit
  REM saleins: (comcheck), linktoys
  REM seemore: (f): continue loop while found()
  REM start: closes all files, opens, sload files, sets f$
  REM toyclear: (toyins), clears field variables
  REM toyinput: (toyins) (comcheck), sinput fields except code$
  REM toyins: (comcheck), append if code$ is new
  REM warning: print warning (msg$) at 14,5
  REM xchange: (comcheck) uses other file, resets f$
  REM zoom: (comcheck), next loop while not eof
  endproc
proc b
  back
  if recnum()=0
    warning; "Start of file"
    endif
  endproc
proc blank; x,y
  print at x,y; rept(" ",64-y+1);
  endproc
proc closeall
  while 1
    close
    endwhile
  endproc
proc comcheck
  if c$="?": help: endif
  if c$="X": xchange: endif
  if c$="S": stop : endif
  if c$="D": dele: endif
  if c$="J": jump: endif
  if c$="R": error reverse: endif
  if c$="1": first : endif
  if c$="L": last : endif
  if c$="Z": error zoom: endif
  if c$="F": f: endif
  if c$="N": n: endif
  if c$="B": b: endif
  if c$="A": error a: errchk: endif
  if c$="I"
    if f$="SALES": saleins: else : toyins: endif
    endif
  endproc
proc comget
  let othercode$=""
  while 1
    if code$<>othercode$
      linkrecs
      endif
    sprint
    blank; 13,5
    print at 12,5; "Command ?: "; paper 6; " "; paper 0; " (Press ? for help)"
```

```
    let c$=upper(getkey())
    blank; 14,5
    print at 12,16; c$
    comcheck
    endwhile
  endproc
proc dele
  prompt; "Confirm delete (y/n)"
  input txt$
  if lower(txt$)="y"
    delete
    endif
  endproc
proc errchk
  if errnum()>0 and errnum()<>27
    warning; "Error "; +str(errnum(),3,0)
    endif
  endproc
proc escmsg
  prompt; "Press ESC to stop"
  endproc
proc f
  prompt; "Find ?: "
  input txt$
  if txt$>""
    find txt$
    if found()
      sprint
      seemore
      else
      warning; txt$+" not found"
      endif
    endif
  endproc
proc gettoycode
  prompt; "Toy code ? "
  input newcode$
  let newcode$=upper(newcode$)
  endproc
proc help
  cls
  print "PRESS ONE OF THE FOLLOWING KEYS TO PERFORM A TASK"
  print "B  go back a record"
  print "N  go to next record"
  print "A  alter the record on the screen"
  print "I  insert a new record"
  print "F  find record(s)"
  print "S  stop"
  print "D  delete current record"
  print "1  go to first record"
  print "L  go to last record"
  print "Z  zoom/fast forwards"
  print "R  reverse/fast backwards"
  print "J  jump a number of records forwards or backwards (minus)"
  print "X  change the current file"
```

```
  print
  print "Press any key to continue"; getkey()
  screen
  endproc
proc jump
  prompt; "Records to jump ?"
  input num
  position recnum()+num
  endproc
proc linkrecs
  let othercode$=code$
  if f$="TOYS"
    linksale; code$
    else
    linktoys; code$
    endif
  endproc
proc linksale; seek$
  use "s"
  locate seek$
  if s.code$<>seek$
    blank; 8,0
    let othercode$=""
    else
    sprint
    endif
  use "t"
  endproc
proc linktoys; seek$
  use "t"
  locate seek$
  sprint
  use "s"
  endproc
proc n
  next
  if eof()
    warning; "End of file"
    endif
  endproc
proc prompt; msg$
  print at 13,5; ink 4; msg$;
  endproc
proc reverse
  escmsg
  while recnum()>0
    b
    sprint
    endwhile
  endproc
proc saleclear
  let month=0
  let qty=0
  let disc=0
  let profit=0
```

```
  sprint
  endproc
proc saleinput
  sinput month,qty,disc
  let profit=qty*(t.price*(1-disc/100)-t.cost)
  append
  sprint
  endproc
proc saleins
  gettoycode
  linktoys; newcode$
  if t.code$=newcode$
    let code$=newcode$
    saleclear
    error saleinput
    errchk
    else
    warning; "No such toy code"
    endif
  endproc
proc seemore
  while found()
    prompt; "Press any key to see more, or S to stop"
    if upper(getkey())="S"
      return
      else
      continue
      sprint
      endif
    endwhile
  warning; " No more "+txt$
  endproc
proc start
  cls
  print at 5,10; ink 2; paper 6; " Opening Files .... "
  error closeall
  open "sales"logical "s"
  order code$; a,month; a
  open "toys"logical "t"
  order code$; a
  let f$="TOYS"
  sload "toysales"
  comget
  endproc
proc toyclear
  let name$=""
  let cost=0
  let price=0
  sprint
  endproc
proc toyinput
  sinput name$,cost,price
  append
  endproc
```

```
proc toyins
  gettoycode
  locate newcode$
  if code$=newcode$
    warning; "Code already in use"
    else
    let code$=newcode$
    toyclear
    error toyinput
    errchk
    endif
  endproc
proc warning; msg$
  print at 14,5; paper 6; ink 2; " "; msg$; " "
  endproc
proc xchange
  if f$="TOYS"
    use "s"
    let f$="SALES"
    else
    use "t"
    let f$="TOYS"
    endif
  endproc
proc zoom
  escmsg
  while not eof()
    sprint
    n
    endwhile
  endproc
```

EXERCISES

Note: Use a special backup for exercises.

1 Alter the `delete` procedure so that it does not delete a toy record if sales records for it still exist.

2 Split the procedure `comcheck` into two separate procedures, with one or the other called from `comget` according to the value of `c$`. This will speed up the program's response to a key press, especially useful with commands like `n` or `b`. ARCHIVE will only have to read half the number of `if c$` lines each time.

3 Change the way records are altered, so that the `code$` field cannot be changed, and to include the calculation of profit on sales records. Use `saleinput` and `toyinput` to do this, coupled with `update`.

SUGGESTED ANSWERS

1 Add a few lines to `dele` to check for a sales record if the current file is `TOYS`:

```
proc dele
  prompt; "Confirm delete (y/n)"
  input txt$
  if lower(txt$)="y"
    if f$="TOYS"
      linksale; code$
      if s.code$=t.code$
        warning; " Sales exist for this toy "
        else
        delete
        endif
      else
      delete
      endif
    endif
  endproc
```

2 Physically split `comcheck` into two procedures (using cut and paste), then check `c$` in `comget` using `instr()` to determine which to call.

```
proc comcheck
  if c$="B": b: endif
  if c$="N": n: endif
  if c$="F": first : endif
  if c$="L": last : endif
  if c$="J": jump: endif
  if c$="?": help: endif
  if c$="D": dele: endif
  if c$="X": xchange: endif
  if c$="F": f: endif
  endproc
```

```
proc comcheck2
  if c$="S": stop : endif
  if c$="R": error reverse: endif
  if c$="Z": error zoom: endif
  if c$="A": error a:errchk:endif
  if c$="I":if f$="SALES": saleins: else : toyins: endif:endif
  endproc
```

Alter the line `comcheck` in `comget` to read:

```
proc comget ......
  if instr("PVAIRZS",c$): comcheck2: else : comcheck: endif
  endproc
```

3 Create new procedures to `append` if `c$` is `I`, or `update` if it is `A`. Call this procedure from `toyinput` and `saleinput` (instead of using `append` directly).

```
proc toyinput
  sinput name$,cost,price
  savechange
  endproc

proc savechange
  if c$="I"
    append
    else
    update
    endif
  endproc
```

```
proc saleinput
  sinput month,qty,disc
  let profit=qty*(t.price*(1-disc/100)-t.cost)
  savechange
  sprint
  endproc
```

In the procedure `comcheck2`: delete `error a` from the line which calls the procedure `a`, and instead add:

```
if c$="A"
    if f$="SALES": error saleinput: else : error toyinput: errchk:
endif
    endif
```

Delete procedure `a`.

Note: Remember to add these model solutions to your working tape.

CHAPTER 12

Listing, Paging and Totalling

Handwritten and typed lists suffer from a greater drawback than writer's cramp and wrist fatigue - if they need changing at all, they probably need to be started from scratch.

ARCHIVE prints the same data in as many different formats, sequences and selections as you require, as often as you want. All you waste is some paper, some ribbon, and a great deal less time than it would take to do manually.

ARCHIVE also cross-refers, calculates, averages, totals and subtotals the printed lists in any way you require, effectively dispensing with hours spent looking up figures and tapping away on a calculator.

This chapter shows how to:

- Set up listing programs which work with either screen or printer
- Determine the type (string or numeric) of a field
- List two linked files
- Stop a display scrolling
- Subtotal and average fields

- The following ARCHIVE commands are introduced:
  `local` – designates a variable as local to the procedure it is used in
  `mode` – changes the screen width, and whether three areas are displayed or not

- The following ARCHIVE functions are introduced:
  `fieldt()` – used to discover whether a field is of the numeric or string type
  `inkey()` – checks to see if a key has been pressed and returns its value if so

Points to remember

- When several loops are nested, the action which occurs most often must be in the deepest loop.
- Totals are calculated using loops just like counters. First they must be zeroed, then incremented, and finally ended when the end of the records to be totalled is reached.

Example: Listing the toy sales

Harold has completed the procedures for inserting, amending and browsing through the toy and sales records.

Most computer programs fall into three distinct parts: input, processing, and output. The processing is the primary purpose for using a computer – tasks such as ordering the data and performing calculations are done faster and more accurately by computer than by hand.

Processing tends to take place both at the time of input (eg. calculating the profit) and output (eg. totalling the sales), so essentially Harold's procedures fall into just two groups: those concerned with input, and those concerned with output.

He now turns his attention to the procedures for output, designed to list the records (with totals) on either the screen or printer.

INSTRUCTIONS

1 `open` the `toys` file with the logical name `t`

2 `open` the `sales` file with the logical name `s`

Both files should already be ordered since they were saved with an order imposed on them.

If the files are already open, save any procedures in memory before proceeding.

3 `use` the `sales` file, if it is not the current file.

List on screen or printer?

Harold decides to use one or two of Miranda's listing procedures as the corner stones of his own program, but to delete those that do not apply.

```
load "lists" ↵
```

1 Load the program file `lists`

2 Delete the procedures `dispguest`, `listguests`, `listmemos`, `prinrec`, `printmemos` and `zoom`, as they are not relevant.

The remaining procedures should be `disprec`, `ff`, `line` and `list`.

Harold's listing procedures will be called from `comget` with a single key press, just like all the other tasks. He allocates the letter `p` for sending the lists to the printer, and `v` for sending them to the screen, or *VDU* (visual display unit). The letter `s` is already in use for `stop`.

Rather than write quite separate sets of procedures for these two different requirements, he adapts `line` to either `print` or `lprint` accordingly.

At the moment, `line` looks like this:

```
proc line; l$
  lprint l$
  endproc
```

3 Make `line` the current procedure.

```
if c$="P" ↵lprint l$; ↵
```

4 Add an `if` clause so that if `c$` (the command variable) is `P`, then the procedure `lprint`s the string passed to `l$`. Otherwise it `print`s it.

Note the semicolon after `l$` to suppress the automatic carriage returns, as the sales will be printed in a tabular format, one record per line, by `disprec`.

```
else ↵print l$; ↵
```

5 Put in an `else` clause to send the information to the screen if `c$` is not `P`. This line also ends in a semicolon.

```
endif
```

6 End the `if` clause.

It should look like this:

```
proc line; l$
  if c$="P"
    lprint l$;
    else
    print l$;
    endif
  endproc
```

Changing the DISPREC procedure

1 Make `disprec` the current, highlighted procedure.

At the moment it looks like this:

```
proc disprec
  let count=0
```

```
while count<numfld()
  print fieldv(count)
  let count=count+1
  endwhile
endproc
```

```
proc listrec
```

2 Change the name of the procedure to `listrec` to clarify its purpose.

```
line;fieldv(count)+" "
```

3 Change the line `print fieldv(count)` in the newly named `listrec` procedure so that it calls the procedure `line` as follows:

```
line;fieldv(count)+" "
```

Note that the space has been concatenated with (added to) the field using a plus sign rather than the semicolon, because `line` must be passed only one single string as a parameter.

```
linecr
```

4 Add a line `linecr` after the loop to call a procedure (described next) to print a carriage return after the record has been printed on one line.

The procedure now looks like this:

```
proc listrec
  let count=0
  while count<numfld()
    line; fieldv(count)+" "
    let count=count+1
    endwhile
    linecr
  endproc
```

The procedure LINECR

```
proc linecr↵
  line; chr(13)+chr(10) ↵
  endproc ↵
```

1 Create the procedure `linecr` as follows:

```
line; chr(13)+chr(10)
endproc
```

The procedure `linecr` sends literal ASCII characters `CR` and `LF` to the printer or screen, since the carriage return has been suppressed in `line` by using semicolons.

Changing LIST

1 Make `list` the current, highlighted procedure.

At the moment it looks like this:

```
proc list
  first
  cls
  while not eof()
    disprec
    next
    endwhile
  endproc
```

`listrec` 2 Change the line `disprec` in the procedure `list` so that it now calls `listrec`.

`ESC` 3 Leave `edit`

4 Save the procedures as `toylists`

`let c$="V"` 5 Type `let c$="V"` to declare the variable and prevent `line` using the printer.

`list` 6 Type `list` to try out the procedures.

The screen clears and starts to list the first record in the current file, but immediately collapses with the error message:

```
Error 72: Type Mismatch
line    : line;l$
```

The procedure `line` expects to be passed a string as a parameter. The field value returned by `fieldv()` is numeric as soon as the second field (`month`) is reached.

The function FIELDT()

The way to overcome this problem and continue to use `listrec` and `line` in the way intended is to convert any numeric fields to strings using the function `str()`.

First, however, you need to be able to tell which fields are numeric variables.

You can use the function `fieldt()` to determine the type (string or numeric) of any field.

Like the function `fieldv()`, `fieldt()` returns a value for the field whose field number appears between the brackets.

For example, `print fieldt(0)` displays a number indicating the type of the first field (field 0) in the current file.

If the field is a character or string field, `fieldt()` returns the value 1. If the field is numeric, it returns the value 0.

Change the procedure `listrec` so that the value returned by `fieldv()` is converted to a general format string if the field is

numeric. Add an `if` clause using `fieldt()` to check when this needs to be done.

Remember that `str()` in the general format includes decimal places if they exist, and not if they don't.

1 Use `edit` and **TAB** to `listrec`

2 Add the following line at the start of the loop:

```
if fieldt(count)
```

Remember that the variable `count` starts at `0` and increments until it equals the number of fields in the record.

The line `if fieldt(count)` is short for `if fieldt(count)=1`. It executes the next command line if the function returns the value `1` (the field is a string field).

3 Insert these lines after the `line` command:

```
else
line; str(fieldv(count),3,0)+" "
endif
```

If the number returned by `fieldt()` is `0` (the field is numeric), the procedure executes the line after `else`. This converts the field's value to a string before adding a space and passing it as a parameter to `line`.

The altered procedure looks like this:

```
proc listrec
  let count=0
  while count<numfld()
    if fieldt(count)
      line; fieldv(count)+" "
      else
      line; str(fieldv(count),3,0)+" "
      endif
    let count=count+1
    endwhile
  linecr
  endproc
```

4 Leave `edit` and type the command `list`.

This time all the records in the current file are listed, one line at a time:

```
LLU 7 525 50 420
LLU 8 475 50 380
LLU 9 330 20 1105.5
LLU 10 260 20 871
```

```
LLU 11 210 0 1060.5
LLU 12 300 50 240
LLZ 1 10 0 76.5
LLZ 2 180 0 1377
LLZ 3 310 0 2371.5
LLZ 4 320 0 2448
LLZ 5 230 0 1759.5
LLZ 6 260 0 1989
LLZ 7 -20 0 -153
LLZ 8 210 0 1606.5
LLZ 9 180 0 1377
LLZ 10 135 0 1032.75
LLZ 11 120 0 918
LLZ 12 90 0 688.5
```

The listing scrolls off the screen, however, preventing the first records from being properly read.

Pausing the listing – first count the lines

Harold decides to create a procedure using `getkey()` to pause during the listing when the screen is full. A single key press allows the procedure to continue.

`edit`

1 Type the command `edit`

2 Create the following procedure:

```
proc pause
  print "Press a key to continue"; getkey()
  endproc
```

The cleared DISPLAY AREA accepts 14 lines of print before starting to scroll. The procedure `pause` will be used when 13 lines have been printed on the screen.

To know when to do this, the program counts the number of lines printed. Once the screen is full, the count will have to be zeroed to start counting for the next screen full.

Add some lines to `linecr` to count the lines and check to see if it's time to pause.

`let lc=lc+1`

3 TAB to the procedure `linecr` and insert the line `let lc=lc+1` at the end. Every time a carriage return is used, a new line starts.

The variable `lc`, short for 'line count', is used to store the number of lines printed. Here it is incremented each time a new line starts.

```
if lc=13 ↵linemax ↵endif ↵
```

4 Insert the following `if` clause:

```
if lc=13
  linemax
  endif
```

When the value of `lc` reaches 13 (the 14th line), the procedure `linemax` (described next) takes whatever action is necessary.

The finished procedure looks like this:

```
proc linecr
  line; chr(13)+chr(10)
  let lc=lc+1
  if lc=13
    linemax
    endif
  endproc
```

Pausing listings with INKEY()

An alternative way of stopping scrolling is to let the listing continue unless a key is pressed, rather than pausing at regular intervals.

The function `inkey()` is like the function `getkey()`, in that it fetches a single character from the keyboard. Unlike `getkey()`, `inkey()` does not hold up the procedure, but looks to see if a key is being pressed.
`inkey()` could be used every time a line is printed to check if you have pressed a key to hold up the listing, eg: `if inkey()>"":pause:endif`

Note that `inkey()` does not work very well in early versions of ARCHIVE as it is not very responsive, so it is not used here.

Taking action when the screen or page is full

1 Create the procedure `linemax`

```
let lc=0
```

2 Type the first line `let lc=0`
This sets the line counter to zero, ready to start counting again.

```
if c$="P" ↵ff ↵
```

3 Now type the two following lines:

```
if c$="P"
  ff
```

This ejects the paper to the top of the next sheet if the printer is being used.

The procedure `ff` already exists, left over from `lists`.

`else ↵ pause ↵` 4 Now type two lines to make the procedure pause:

```
else
pause
```

If the value of `c$` is not `P`, (the listing is on the screen), this pauses the program.

`endif` 5 Finish typing the procedure with an `endif` line.

The finished procedure looks like this:

```
proc linemax
  let lc=0
  if c$="P"
    ff
    else
    pause
    endif
  endproc
```

`if lc=lmax` 6 Change the number 13 in the procedure `linecr` to the variable `lmax` with `if lc=lmax`

This allows for a different number of maximum lines according to whether the listings are to be printed or shown on screen.

A typical number of lines on continuous (fanfold) computer printout paper is 66 in total, so a figure of 55 leaves suitable margins top and bottom.

Note: different display modes in ARCHIVE (with or without the WORK AREA and the CONTROL AREA) will also require different maximum line settings. They can be changed using the command `mode`.

The procedure now looks like this:

```
proc linecr
  line; chr(13)+chr(10)
  let lc=lc+1
  if lc=lmax
    linemax
    endif
  endproc
```

The command MODE

`mode` changes the number of columns displayed on the screen, and also switches the CONTROL and WORK AREAS on or off.

`mode` must be followed by two numbers (or expressions) separated by commands.

If the first number (or argument) is 1, the boxed areas are on: if it is 0, they are switched off.

There are 14 lines of display with the areas switched on, and 24 when they are off. Commands and display appear after each other when there is no WORK AREA.

Note: a bug in early versions of ARCHIVE causes `edit` to crash when used with no WORK AREA, and more than 19 procedures.

The second number uses the first figure of the three possible screen widths (40, 64 and 80) to designate which should be used.

Examples:

`mode 1,6`	sets 3 areas and 64 columns (like pressing F2 to load ARCHIVE).
`mode 1,8`	sets 3 areas and 80 columns (like pressing F1 to load ARCHIVE).
`mode 0,4`	sets no areas and 40 columns.

When only the CONTROL AREA is switched off (using the F2 key), there are 19 lines in the DISPLAY AREA.

Declaring temporary working values

1 Create a procedure called `setup` to be used temporarily to set up the starting values for the variables needed by these procedures.

When they are used later in a computer program, they will be set by other procedures, and `setup` will become redundant.

```
let c$="V"
```

2 Type the following line to set up the variable `c$` which tells it to send lists to either the screen or the printer (`"V"` sends them to the screen):

```
let lc=0 ↵ let lmax=13 ↵ let c$="V"
```

3 Set the variable `lc` to 0, and `lmax` to 13:

```
let lc=0
let lmax=13
```

`list` 4 Make `setup` call the `list` procedure.

The procedure should look like this:

```
proc setup
  let c$="V"
  let lc=0
  let lmax=13
  list
  endproc
```

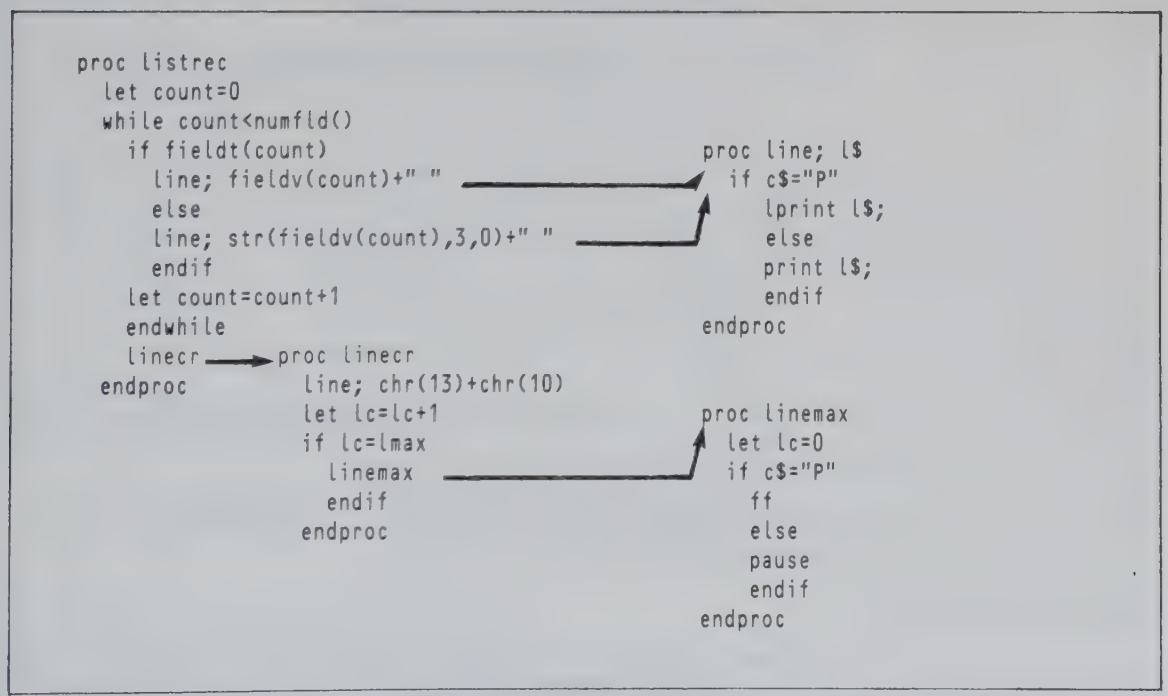
5 Leave `edit` and ...

6 save the procedures as `toylist2`.

`setup` 7 Type in `setup` to set the variables, and to see the results.

The records appear on the screen until nearly at the bottom, then the program stops and the `pause` message appears. Press a key to continue with the listing.

The procedures work together like this:



Totalling the profits

Harold is approaching the ultimate goal of all his endeavours: subtotalling and comparing the profits of Lizards and Llamas.

He adds a few lines to his procedures to use an *accumulator* to hold a running (accumulating) total of the profits. This increments as each record is printed, and the result is printed at the end.

A running total works just like a counter in a loop. It starts as a zero value variable before the loop, then each passing record adds its value to the variable until the loop is finished.

For example, the following line totals the `profit` field in the `sales` file:

```
let tot=0:all "s":let tot=tot+s.profit:
endall:print tot
```

1 Call up the procedures in memory for editing and make `list` the current, highlighted procedure.

2 Insert the line `let tot=0` before the loop in `list`. This zeros the accumulator.

3 Insert this line inside the loop:
`let tot=tot+profit`
This increments the accumulator.

4 Insert the following line after the loop:

```
line; "* TOTAL PROFIT: "+str(tot,3,0)
```

to print the accumulated total.

The procedure looks like this after the changes:

```
proc list
  first
  cls
  let tot=0
  while not eof()
    listrec
    let tot=tot+profit
    next
    endwhile
  line; "* TOTAL PROFIT: "+str(tot,3,0)
  endproc
```

5 Leave `edit` and type `setup` to test the totalling.

Subtotalling by toy

Subtotals work in exactly the same way, but are printed out and then zeroed again each time the group being subtotalled changes.

Harold creates a new procedure to list and subtotal sales only as long as they have the same code. Since they are ordered on `code$`, this can easily done by storing the starting code and using a loop until the current code is different.

1 Use the `edit` command to call up the procedures and make `list` the current, highlighted procedure.

2 Insert the line `let lastcode$=code$` at the top of the loop.
This stores away the current value of `code$`.

3 Change the line `listrec` to `listsales`
This calls a procedure which lists the records as long as `code$` is the same as `lastcode$`.

4 Change the line `let tot=tot+profit` to read:

```
let tot=tot+subtot
```

The variable `subtot` is accumulated in `listsales`, and added to `tot` each time it finishes.

5 Cut the line `next` from the procedure `list`
When more than one loop is in use, the `next` command must be performed in the deepest loop, in this case, `listsales`.

The procedure now looks like this:

```
proc list
  first
  cls
  let tot=0
  while not eof()
    let lastcode$=code$
    listsales
    let tot=tot+subtot
    endwhile
  line; "* TOTAL PROFIT: "+str(tot,3,0)
  endproc
```

Listing and totalling sales for one toy

1 Use `edit` to create the procedure `listsales`.

`let subtot=0`

2 The first line should read: `let subtot=0`.
This line zeroes the variable `subtot`, the accumulator for the subtotal.

`while not eof() and code$=lastcode$`

3 The next line `while not eof() and code$=lastcode$` begins a loop which continues as long as the `code$` in the sales record stays the same as `lastcode$`.
Note that `not eof()` appears in both this loop and overall loop in `list`. Without it, the program will stick in this loop endlessly printing the last record, which will have the value of the last `lastcode$`

`listrec`

4 Type a line to call the `listrec` procedure.

`let subtot=subtot+profit`

5 The next line, `let subtot=subtot+profit` increments the accumulator.

`next` 6 Now type `next`.
Remember that `next` must be in the deepest loop when loops are nested.

`endwhile` 7 Close the `while` loop with the line `endwhile`.

8 Type the line:

```
line; "* PROFIT for "+lastcode$+": "
+str(subtot,3,0)
```

which prints the subtotal.

`linecr` 9 Now type `linecr`
This starts a new line after printing the subtotal, and increments the line counter.

The finished procedure should look like this:

```
proc listsales
  let subtot=0
  while not eof() and code$=lastcode$
    listrec
    let subtot=subtot+profit
    next
    endwhile
  line; "* PROFIT for "+lastcode$+": "+str(subtot,3,0)
  linecr
  endproc
```

10 Leave `edit`.

11 Try out these listing procedures by calling the procedure `setup`.

```
LLU 7 525 50 420
LLU 8 475 50 380
LLU 9 330 20 1105.5
LLU 10 260 20 871
LLU 11 210 0 1060.5
LLU 12 300 50 240
* PROFIT for LLU: 4077
LLZ 1 10 0 76.5
LLZ 2 180 0 1377
```

```
LLZ 3 310 0 2371.5
LLZ 4 320 0 2448
LLZ 5 230 0 1759.5
LLZ 6 260 0 1989
LLZ 7 -20 0 -153
LLZ 8 210 0 1606.5
LLZ 9 180 0 1377
LLZ 10 135 0 1032.75
LLZ 11 120 0 918
LLZ 12 90 0 688.5
* PROFIT for LLZ: 15491.25
* TOTAL PROFIT: 19568.25
```

Averaging the sales

The subtotals displayed are not very meaningful, because Luny Llama has only been on sale for six months, whereas Lawrence Lizard has been going for a year.

Harold adds one last counter to count the number of sales records for each toy so that the program calculates the average profit.

1 Use `edit` and make `listsales` the current, highlighted procedure.

2 Insert `let count=0` before the loop

3 Insert `let count=count+1` in the loop

4 Insert after the loop:

```
line; "* AVERAGE: "+str(subtot/count,3,0)
linecr
```

The average monthly profit is the value of the subtotalled profits divided by the number of months.

The procedure now looks like this:

```
proc listsales
  let subtot=0
  let count=0
  while not eof() and code$=lastcode$
    listrec
    let subtot=subtot+profit
    let count=count+1
    next
    endwhile
  line; "* PROFIT for "+lastcode$+": "+str(subtot,3,0)
  linecr
```

```
line; "* AVERAGE: "+str(subtot/count,3,0)
linecr
endproc
```

Local variables

Note that the variable `count` has the same name as the counter variable in `listrec`, which is called by `listsales` once for every sales record with the same toy code.

As it stands, `listrec` will override the value of `count` in `listsales`, leaving it with the value of the number of fields in the record plus 1. This obviously ruins the chances of accurately counting the number of records.

To overcome this problem, use the command `local` in the procedure `listrec`.

`local count`

1 TAB to the procedure `listrec` and insert the line `local count` at the start of the procedure.

The procedure now looks like this:

```
proc listrec
  local count
  let count=0
  while count<numfld()
    if fieldt(count)
      line; fieldv(count)+" "
      else
      line; str(fieldv(count),3,0)+" "
      endif
    let count=count+1
    endwhile
    linecr
  endproc
```

2 Leave `edit` and

3 `save` the procedures to `toylist3`.

`setup`

4 Type `setup` to check that the procedures work:

```
LLU 7 525 50 420
LLU 8 475 50 380
LLU 9 330 20 1105.5
LLU 10 260 20 871
LLU 11 210 0 1060.5
LLU 12 300 50 240
* PROFIT for LLU: 4077
* AVERAGE: 679.5
```

```
LLZ 1 10 0 76.5
LLZ 2 180 0 1377
LLZ 3 310 0 2371.5
LLZ 4 320 0 2448
LLZ 5 230 0 1759.5
LLZ 6 260 0 1989
LLZ 7 -20 0 -153
LLZ 8 210 0 1606.5
LLZ 9 180 0 1377
LLZ 10 135 0 1032.75
LLZ 11 120 0 918
LLZ 12 90 0 688.5
* PROFIT for LLZ: 15491.25
* AVERAGE: 1290.9375
* TOTAL PROFIT: 19568.25
```

The Command LOCAL

The command `local` makes any variable name that follows it local to the procedure `local` is used in.

These are called *local* (as opposed to *global*) variables, and anything that happens to them in the procedure is ignored by all other procedures in the program.

Variables with the same name in other procedures are unaffected by variables local to procedures that they call, or are called by.

In the case of `listsales` and `listrec`, the value of `count` varies as follows when there are (for example) three sales with the same toy code:

In procedure:	Value of `count`:
`listsales`	:0
`listrec`	:0,1,2,3,4,5 (first record)
`listsales`	:1
`listrec`	:0,1,2,3,4,5 (second)
`listsales`	:2
`listrec`	:0,1,2,3,4,5 (third)
`listsales`	:3

Linking the files

Harold now starts to consider linking the two files together in the listings, and merging the `toylists` procedures with those in `toys`.

First he alters the main listing procedure to make `toys` the current data file and list each toy before moving on to `listsales`.

He then changes `listsales` so that it uses the `sales` file throughout, and finds sales records linked with the current toy.

1 Use `edit` and make `list` the current, highlighted procedure.

```
use "t" ↵ let f$="TOYS" ↵
```

2 Insert these two lines at top of the procedure to make sure that the `toys` file is the current file when the listings start:

```
use "t"
let f$="TOYS"
```

`listrec` 3 Insert `listrec` at the top of the loop to print the current toy record before going on to list linked sales.

4 Delete the line `let lastcode$=code$`, as the last code will now be `code$` in the toys file.

`next` 5 Type `next` which in this case affects the toys file only.

The changed procedure now looks like this:

```
proc list
  use "t"
  let f$="TOYS"
  first
  cls
  let tot=0
  while not eof()
    listrec
    listsales
    let tot=tot+subtot
    next
    endwhile
  line; "* TOTAL PROFIT: "+str(tot,3,0)
  endproc
```

6 **TAB** to the procedure `listsales`

`use "s"` 7 Insert the line `use "s"` at the start of the procedure.

The `sales` file is the current file throughout this procedure.

`locate t.code$` 8 Type `locate t.code$`
If there are any sales records for the current toy, `locate` finds the first one.

If there are none, the loop fails before starting, since the sales `code$` is not the same as `t.code$`.

`t.code$` 9 Change `lastcode$` to `t.code$` in the rest of the procedure.

```
use "t"
```

10 Insert the line `use "t"` at the end of the procedure to make `toys` the current file again.

The procedure now looks like this:

```
proc listsales
  use "s"
  locate t.code$
  let subtot=0
  let count=0
  while not eof() and code$=t.code$
    listrec
    let subtot=subtot+profit
    let count=count+1
    next
    endwhile
  line; "* PROFIT for "+t.code$+": "+str(subtot,3,0)
  linecr
  line; "* AVERAGE: "+str(subtot/count,3,0)
  linecr
  use "t"
  endproc
```

11 Delete the procedure `setup`, as it will not be needed once the variables are set up in procedures like `comget` and `comcheck`.

12 Leave `edit` and

13 save the procedures to the program file `toylists`.

Merging the two programs

Harold is now ready to merge the procedures from `toys` and create a procedure to link the two sets together.

```
merge "toys"
```

1 Type `merge "toys"` to create one large program with all the input and output procedures.

```
edit
```

2 Type `edit` and **TAB** to the procedure `comcheck`

```
if c$="P":let lmax=55:listselect:endif ↵
if c$="V":let lmax=14:listselect:endif ↵
```

3 Insert two lines to act on the command keys `P` (for printer) and `V` (for VDU)

Note that both lines call the same procedure, but with different values of `lmax`.

```
linecr ↵ pause ↵
```

4 **TAB** to the procedure `list` and insert these two lines at the bottom of the procedure.

This ensures a pause at the end of the listing so that the totals can be read on the screen before the program returns to `comget`.

```
use "t" ↵let f$="TOYS" ↵first ↵cls ↵
```

5 Cut these four lines from the procedure so they can be pasted into a new one.

Be careful not to clear the cut and paste buffer, as these are about to form the basis of a new procedure.

The procedure now looks like this:

```
proc list
  let tot=0
  while not eof()
    listrec
    listsales
    let tot=tot+subtot
    next
    endwhile
  line; "* TOTAL PROFIT: "+str(tot,3,0)
  linecr
  pause
  endproc
```

6 Create the procedure `listselect`
This procedure will be used to get the starting values and the range of records needed before listing.

Remember to press ESC to leave insert mode.

```
use "t" ↵let f$="TOYS" ↵first ↵cls ↵
```

7 Paste in the lines cut from `list`.

```
let lc=0 ↵list ↵screen ↵
```

8 Insert these three lines to set the line counter, call the listing procedure, and then redisplay the screen layout afterwards.

The finished procedure looks like this:

```
proc listselect
  use "t"
  let f$="TOYS"
  let lc=0
  first
  cls
  list
  screen
  endproc
```

9 Update the `a1` procedure to take account of all the new procedures. Add the following lines:

```
REM ff: (linemax) sends form feed to printer
REM line: (listrec) prints l$ param if c$="V" or lprints if "P"
REM linecr: (listrec) CR and LF as param to line
REM linemax: (linecr) form feed for printer, pause for screen
REM list: (listselect) controlling loop, prints total profit
REM listrec: (listsales) lists every field one by one on one line
REM listsales: (list) sales loop for one toy, prints subtot, average
REM listselect: (comcheck) gets start vals and range for listing
REM pause: (linemax) getkey to continue listing
```

10 Update the `help` procedure to include the new commands `P` and `V`.

11 Leave `edit` and

12 save the combined procedures to a file called `fulltoys`.

Selecting and listing a range of records

Harold has one final enhancement he wants to make to the listing programs: the ability to list only part of the files, rather than every record every time.

Choosing a record to start from is easy. He adds lines to prompt for the starting toy code, and then locates it before starting the listing.

Choosing a record to finish with is pretty straight forward as well. He clears the prompt for the start record, prompts for an end record, and then adds an expression to the file loop so that it continues only while the current code is less than or equal to the chosen final code.

All codes in the range between the starting and the finishing code will be printed.

Processing a range of records like this is simple when a file is ordered on the field chosen to designate the range.

`edit` 1 Edit the procedure `listselect`

`prompt;"From code?" ↵input from$ ↵`

2 Insert the following lines at the start of the procedure to fetch the letter(s) to start listing from:

```
prompt; "From code ? "
input from$
```

`locate` will be used to find the start, so typing in `A` here starts at the first record, whether it begins with `A` or not.

Similarly, `LL` starts with the first record beginnning with `LL` or higher.

```
if from$>""
```

3 Type the next line: `if from$>""`.
If you find yourself at the `From` prompts because you pressed `P` or `V` accidentally, simply press **ENTER**. This leaves `from$` as a null string, and no further action is taken. The procedure returns control to `comget`.

```
let from$=upper(from$) ↵blank; 13,0 ↵prompt; "To code ?" ↵input
tocode$ ↵let tocode$=upper(tocode$) ↵
```

4 If `from$` is not a null string, these lines blank the prompt line to prompt for the finishing code, and convert the chosen starting and ending codes to upper case.

```
locate from$
```

5 Insert the line `locate from$` after the lines which `use` the `toys` file.

The starting record must be the current record before listing begins.

6 Cut the line `first` for the same reason.

```
endif
```

7 Insert the line `endif` at the end of the procedure to close the `if from$>""` clause.

The procedure now looks like this:

```
proc listselect
  prompt; "From code ? "
  input from$
  if from$>""
    let from$=upper(from$)
    blank; 13,0
    prompt; "To code ? "
    input tocode$
    let tocode$=upper(tocode$)
    use "t"
    let f$="TOYS"
    locate from$
    let lc=0
    cls
    list
    screen
    endif
  endproc
```

```
while not eof() and code$<=tocode$
```

8 **TAB** to the procedure `list` and add the expression `and code$<=tocode$` to the end of the `while` line, so that it reads:

```
while not eof() and code$<=tocode$
```

This will cause the listing to continue until a code higher than the chosen finishing code or the end of the file is reached.

The procedure `list` now looks like this:

```
proc list
  let tot=0
  while not eof() and code$<=tocode$
    listrec
    listsales
    let tot=tot+subtot
    next
    endwhile
  line; "* TOTAL PROFIT: "+str(tot,3,0)
  linecr
  pause
  endproc
```

ESC 9 Leave `edit` and

`save "fulltoys"` 10 Save the procedures to `fulltoys`.

`start` 11 Type in `start` to see the whole program working:

```
LLU Luny Lama 3.45 8.5
LLU 7 525 50 420
LLU 8 475 50 380
LLU 9 330 20 1105.5
LLU 10 260 20 871
LLU 11 210 0 1060.5
LLU 12 300 50 240
* PROFIT for LLU: 4077
* AVERAGE: 679.5
LLZ Lawrence Lizard 2.34 9.99
LLZ 7 -20 0 -153
LLZ 8 210 0 1606.5
LLZ 9 180 0 1377
LLZ 10 135 0 1032.75
LLZ 11 120 0 918
LLZ 12 90 0 688.5
* PROFIT for LLZ: 5469.75
* AVERAGE: 911.625
* TOTAL PROFIT: 9546.75
```

Formatting numeric output

At the moment, Harold's lists are functional but not very pretty. They would look better if all the numbers were lined up in columns of the same width, and the figures were right-justified (so that, for example, the figure 9 in the month column appeared above the 0 of 10 rather than the 1).

This can be done easily in ARCHIVE version 2 using the function `gen()`.

Converting and formatting numbers

ARCHIVE version 2 includes three functions which fulfil a similar task to that of the function `str()`, except that the length of the resulting string can be fixed, and the numbers are ranged right (the normal way of representing numbers in columns).

`GEN()`, meaning general format, is similar to `str()` with 3 as the second argument,

eg: `gen(2*4,10)` returns " 8"

`DEC()`, meaning decimal format, is similar to `str()` with 0 as the second argument,

eg: `dec(13.56897,3,15)` returns " 13.568"

`NUM()`, meaning whole number (integer) format, is similar to `str()` with 2 as the second argument,

eg: `num(23,10)` returns " 23"

The function GEN()

The function `gen()` is similar to the function `str()` with the number 3 (for general format) as its middle argument. It converts any number to a string in the *general* format, that is, whatever it happens to be, without cutting off, forcing or limiting the number of decimal places.

The difference is that `gen()` can determine the overall length of the string (including as many leading spaces as you require), and the number then appears on the right hand side of the string.

`gen()` requires two numeric arguments inside the brackets. The first is the number to be converted, and the second is the overall length of the string.

1 Edit the procedure `listrec`

```
line;gen(fieldv(count)),10)+" "
```

2 Change the second `line` line so that it reads:

```
line;gen(fieldv(count)),10)+" "
```

The procedure now looks like this:

```
proc listrec
  local count
  let count=0
  while count<numfld()
    if fieldt(count)
```

```
        line; fieldv(count)+" "
        else
        line; gen(fieldv(count),10)+" "
        endif
      let count=count+1
      endwhile
    linecr
    endproc
```

3 Leave `edit` and type in `comget` followed by `v` to try it out.

SUMMARY

Harold uses Miranda's `lists` program so that he can list his sales details on either the screen or the printer. He uses the function `fieldt()` to say whether the field to be printed is numeric or text, and then uses the function `str()` to convert the field to text if it is numeric. He creates another procedure which pauses the listing if it is on the screen, or ejects the paper if it is on the printer whenever a screen or page is full.

To complete his program, Harold adds procedures to total, subtotal and average the figures for each toy, so he can then compare them.

Between them Harold and Miranda have created a set of procedures which can be easily adapted to handle the maintenance and listing of any file.

Create your own files and screen layouts, make a few changes to the `fulltoys` program (saved under a new name), and the same procedures will work in exactly the same manner to let you find, browse, insert, amend and list records.

Helpful hint

- Make procedures as general as possible, using variables, so they can be adapted quickly and easily to run with any files.

Program listing

The `fulltoys` program should contain the following new or updated procedures, representing the procedures created in `toylists`:

```
proc a1
  rem      TOYS_PRG  12 May 1984
  REM a: (comcheck), (error) alter
```

```
  REM b: (comcheck), back, warning if start of file
  REM blank: (comget) (linksales), blanks rest of line at co-ordinate x,y
  REM closeall: (comget), closes all open files
  REM comget: controlling loop menu, sprints current, getkey command c$
  REM comcheck: (comget), uses if to test c$
  REM dele: (comcheck), deletes after confirm (if no sales when toy)
  REM errchk: errnum() warning
  REM escmsg: prints escape message
  REM f: (comcheck), input and find txt$, calls seemore
  REM ff: (linemax), sends form feed to printer
  REM first: (comcheck), jumps to first record in file
  REM gettoycode: (toyins) (saleins), prompts for upper (toycode)
  REM help: (comcheck), lists available options, cls and screen
  REM i: (comcheck as error) insert
  REM jump: (comcheck), input num, position recnum()+num
  REM last: (comcheck), goes to last record in file
  REM line: (listrec), prints l$ param if c$="P", lprints if "V"
  REM linecr: (listrec), line
  REM linemax: (linecr), form feed for printer, pause for screen
  REM linkrecs: (comget), locate code$ in other file
  REM linksales: (linkrecs), locates t.code$ in sales or blanks
  REM if othercode$=""
  REM linktoys: (linkrecs) (saleins), locates s.code$ in toys
  REM list: (listselect), controlling loop, prints total profit
  REM listrec: (listsales), lists every field one by one on one line
  REM listsales: (list), sales loop for one toy, prints subtot,average
  REM listselect: (comcheck), gets start vals and range for listing
  REM n: (comcheck), next, warning if end of file
  REM pause: (linemax), getkey to continue listing
  REM prompt: standard proc to print a prompt (msg$) at 13,5
  REM reverse: (comcheck), skips backwards through file
  REM saleclear: (saleins), clears field varibles
  REM saleinput: (saleins), uses SINPUT to put data in fields, calcs profit
  REM saleins: (comcheck), linktoys
  REM seemore: (f): continues to find records until no more
  REM start: closes all files, opens and displays guests
  REM toyclear: (toyins), clears field variables for input
  REM toyinput: (toyins) (comcheck),SINPUT field except code$
  REM toyins: (comcheck), APPEND if code$ is new
  REM warning: standard proc to print warning (msg$) at 14,5
  REM zoom: (comcheck), next loop while not eof
  endproc
proc comcheck2
  if c$="P": let lmax=55: listselect: endif
  if c$="V": let lmax=14: listselect: endif
  if c$="Z": error zoom: endif
  if c$="A"
    if f$="SALES": error saleinput: else : error toyinput: errchk: endif
    endif
  if c$="I": error i: errchk: endif
  if c$="R": error reverse: endif
  if c$="S": stop : endif
  endproc
proc comget
  let othercode$=""
```

```
  while 1
    if code$<>othercode$
      linkrecs
      endif
    sprint
    blank; 13,5
    print at 12,5; "Command ?: "; paper 6; " "; paper 0; " (Press ? for help)"
    let c$=upper(getkey())
    blank; 14,5
    print at 12,16; c$
    if instr("PVAIRZS",c$): comcheck2: else : comcheck: endif
    endwhile
  endproc
proc ff
  lprint chr(12);
  endproc
proc line; l$
  if c$="P"
    lprint l$;
    else
    print l$;
    endif
  endproc
proc linecr
  line; chr(13)+chr(10)
  let lc=lc+1
  if lc=lmax
    linemax
    endif
  endproc
proc linemax
  let lc=0
  if c$="P"
    ff
    else
    pause
    endif
  endproc
proc list
  let tot=0
  while not eof() and code$<tocode$
    listrec
    listsales
    let tot=tot+subtot
    next
    endwhile
  line; "* TOTAL PROFIT: "+str(tot,3,0)
  linecr
  pause
  endproc
proc listrec
  local count
  let count=0
  while count<numfld()
    if fieldt(count)
```

```
      line; fieldv(count)+" "
      else
      line; str(fieldv(count),3,0)+" "
      endif
    let count=count+1
    endwhile
  linecr
  endproc
proc listsales
  use "s"
  locate t.code$
  let subtot=0
  let count=0
  while not eof() and code$=t.code$
    listrec
    let subtot=subtot+profit
    let count=count+1
    next
    endwhile
  line; "* PROFIT for "+t.code$+": "+str(subtot,3,0)
  linecr
  line; "* AVERAGE: "+str(subtot/count,3,0)
  linecr
  use "t"
  endproc
proc listselect
  prompt; "From code ? "
  input from$
  if from$>""
    let from$=upper(from$)
    blank; 13,0
    prompt; "To code ? "
    input tocode$
    let tocode$=upper(tocode$)
    use "t"
    let f$="TOYS"
    cls
    locate from$
    let lc=0
    list
    screen
    endif
  endproc
proc pause
  print "Press a key to continue"; getkey()
  endproc
```

CHAPTER 13

Output Plain and Fancy

This chapter demonstrates:

- how to divert printer output to a file on tape, for use by QUILL, ABACUS and EASEL
- How to divert printer output to the screen
- How to manipulate these facilities to change the structure of a file
- How to list a file with a single command
- How to print a row of labels
- How to create a program (as opposed to a procedure) menu
- How to emulate arrays in ARCHIVE
- How to create procedures which write procedures

- The following ARCHIVE commands are introduced:

`dump`	– produces a one-record-per-line listing of a file
`spoolon`	– diverts printed output to the screen or a file
`spooloff`	– cancels `spoolon`
`export`	– creates data files suitable for use by other programs
`import`	– creates data files from files created by other programs
`save object`	– saves procedures in a tokenised form
`load object`	– loads a tokenised program file

Points to remember

- When using nested loops and testing for a condition which should halt any of them (like `eof()`), remember to repeat the expression testing this condition with every `while`.

- When nesting loops which step through the records of a file, make sure the `next` command goes inside the deepest loop, and don't repeat it in the outer loop.
- ASCII files end with ASCII character number 26 (which must be preceded by ASCII character 0) when creating a file with `spoolon` and `lprint`.
- When printing literal quotation marks as text in spooled files, the combination `"'"` works, while `'"'` doesn't.

Example: Harold puts the boot in

Harold is delighted with Lawrence Lizard's redeeming profit performance, and immediately sets about preparing a report complete with graphs to present to his boss.

He plans to write the report with QUILL and draw the graphs with EASEL. Rather than enter the figures all over again he uses ARCHIVE to create files which QUILL and EASEL can use.

If you have just started ARCHIVE, use `look` to open `toys` (as logical `t`) and `sales` (as logical `s`).

`use "t"` ↵ `cls` ↵ 1 Make toys the current file and clear the screen if necessary.

F2 2 Press F2 to clear the CONTROL AREA
If you do not do this, the next command prints over it making the display hard to read.

`export "_scr" quill`

3 The command `export` followed by `quill` tells ARCHIVE to copy the records of the current file to another file in a format suitable for use by QUILL.

The file is being `exported` so that QUILL can `import` it as a document (text) file.

The DISPLAY AREA shows:

```
code$ name$              cost       price*
LLU   Luny Llama         3.45         8.5*
LLZ   Lawrence Lizard    2.34        9.99*
PUD   Pudgey Penguin     1.78         5.5*
*
```

This shows on the screen (`_scr`) what would have been copied to tape if a filename other than `_scr` had been used. This is how the text appears when imported into QUILL.

Note: The symbol at the end of each line does not appear in QUILL, but denotes the end of line code used internally by QUILL.

The _scr file

The file name _scr is a special file name reserved by the QL to designate the screen. Note that the file name consists solely of the extension (following the underline). Instead of copying toys in its export format onto another tape file, the output has been redirected to the screen.

The screen is obviously not a file on tape, but is here being used as a *logical* file rather than a real one.

As far as the QL is concerned, it uses a similar process whenever it sends out data from memory, whether the output is sent to the screen, a printer, or the Microdrives. The same data can be redirected to any 'foreign' device by simply changing the rules slightly.

The rules for communicating with different devices are kept in programs called drivers. The data in printer_dat, for example, is used by the printer driver.



Harold trying to impress

Exporting to QUILL

To actually create an export file for use by QUILL, type:

```
export "toyfigs" quill
```

1 This creates a file called toyfigs_exp on drive 2 containing the sales file in a text format (as previously displayed).

It can be imported into QUILL by loading QUILL and pressing F3, F, I and entering toyfigs.

It can then be incorporated as part of another document, in this case a report.

Exporting to EASEL and ABACUS

The procedure for creating copies of data files in a format suitable for use by EASEL and ABACUS is exactly the same, except `quill` is ommitted from the command.

```
export "toy-
     graph"
```

1 This creates a file called `"toygraph_exp"`. It can be `im-ported` into ABACUS or EASEL by loading one of these, pressing F3, `F`, `I` and entering the full file name.

```
export "_scr"
```

2 To see on the screen how the exported data is formatted, type in:

```
export "_scr"
```

This diverts the export output away from the Microdrive to the screen.

The DISPLAY AREA shows:

```
"code$","name$","cost","price"
"LLU","Luny Llama",3.45,8.5
"LLZ","Lawrence Lizard",2.34,9.99
"PUD","Pudgey Penguin",1.78,5.5
```

The common format

The display on the screen shows the way that EASEL and ABACUS need to receive data.

The first line contains the field names, in quotation marks, followed by the records each on a new line. Each field is delimited (divided from the next) by a comma, and string fields are in quotes.

The symbol at the end of the list represents here the end of file marker, ASCII character 26.

This is also exactly the format ARCHIVE needs to `import` files. You can `export` files from ABACUS and EASEL, or create them using SUPERBASIC or even ARCHIVE itself, and `import` them into ARCHIVE as data files as long as they follow this format.

Exporting `sales` to EASEL

Data files may not always be constructed in a way suitable for producing graphs. The `sales` file is a good example.

```
use "s"
```

1 Make `sales` the current file and

```
export "_scr"
```

2 display the export format on the screen:

```
"code$","month","qty","discount","profit"
"LLU",7,525,50,420
"LLU",8,475,50,380
"LLU",9,330,20,1105.5
"LLU",10,260,20,871
"LLU",11,210,0,1060.5
"LLU",12,300,50,240
"LLZ",1,10,0,76.5
"LLZ",2,180,0,1377
"LLZ",3,310,0,2371.5
"LLZ",4,320,0,2448
"LLZ",5,230,0,1759.5
"LLZ",6,260,0,1989
"LLZ",7,-20,20,-153
"LLZ",8,210,0,1606.5
"LLZ",9,180,0,1377
"LLZ",10,135,0,1032.75
"LLZ",11,120,0,918
"LLZ",12,90,0,688.5
```

When imported into EASEL as a bar chart, this would produce a graph where the code letters become the cell names for four sets of figures (`month`, `qty`, `discount`, `profit`):

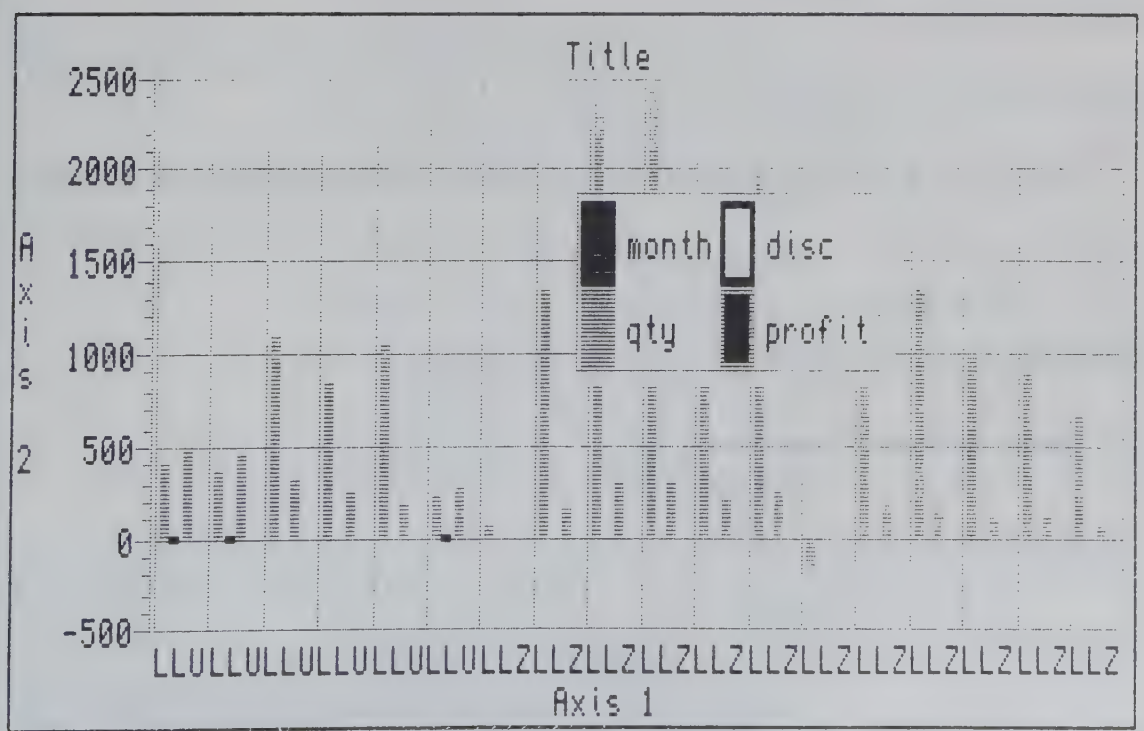


Figure 13.1 – A Bad Export to EASEL

ASCII Files

ASCII files contain only printable ASCII characters (32–127) and the carriage return codes (13 and 10). The files must finish with an end of file marker (26). See the Appendices for more information on ASCII codes.

Program and export files are ASCII files. Screen files and data files (which contain a lot of additional information about the file for ARCHIVE's use) are not.

You can `lprint` other ASCII characters (for `spoolon` files or printer control) by preceding each with an ASCII null, eg: `lprint CHR(0)+chr(26)`. If you do not, they are filtered out by the printer driver.

This is not much use as it stands. For a start, there is obviously not much point in showing the month number as a graph. The real figure of interest is the profit, and the point of the exercise is to compare the toys.

```
select month>6 and code$(to 2)="LL"
```

3 Limit the file to months where figures exist for the two toys.

```
order month;a,code$;a
```

4 Order the file primarily by month.

```
export "_scr";code$,profit
```

5 Export only two fields to the screen.

Note the semicolon before the fields, and the comma separating them.

This displays:

```
"code$","profit"
"LLU",420
"LLZ",-153
"LLU",380
"LLZ",1606.5
"LLU",1105.5
"LLZ",1377
 ..
 ..
etc
```

EASEL would turn this into a more meaningful display as shown in figure 13.2. It is still not perfect as the bars for the two toys are of the same shade, and the months are not printed but Harold can amend these by hand when he prints the graph.

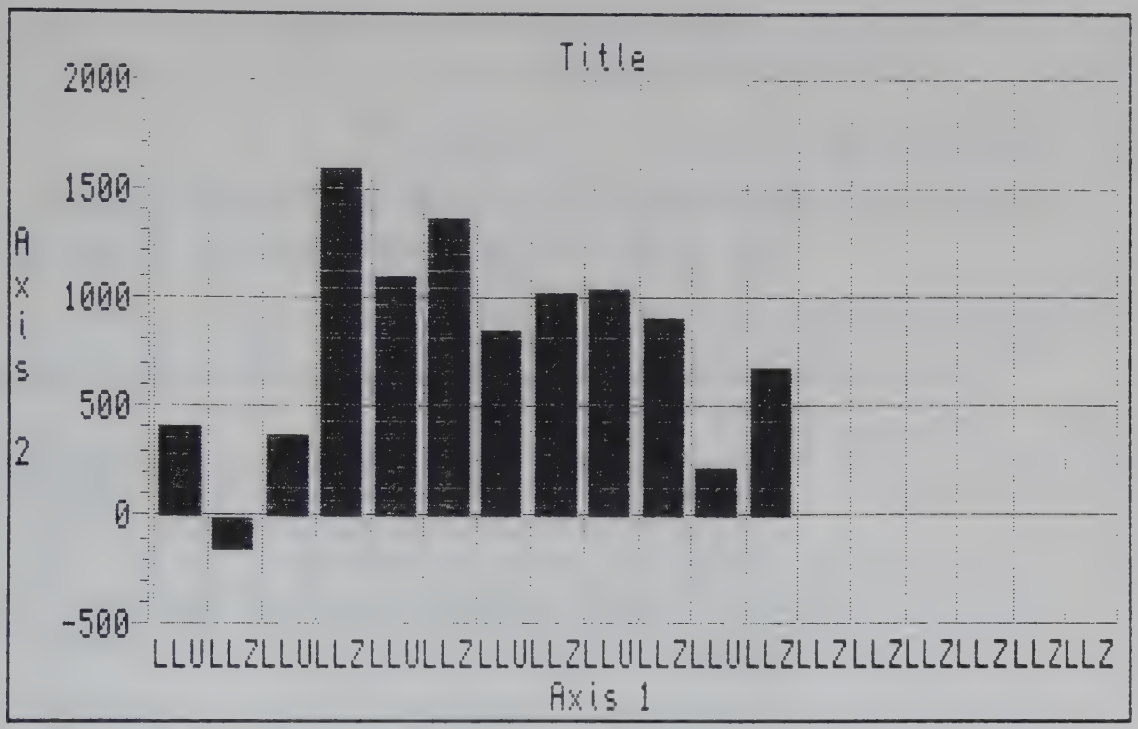


Figure 13.2 – A Better Export to EASEL

```
F5 export "salegraf";code$,profit
F5 export "salefigs";code$,profit quill
```

6 Repeat and edit the line twice to create actual files on tape, suitably formatted for EASEL and QUILL.

Note the space before `quill` in the second command.

```
reset ↵ order code$;a,month;a ↵
```

7 Return the file to its original state.

When exporting from ARCHIVE to EASEL remember:

- The fields (after the first) become sets of figures in the graph (its contents) if numeric.
- The contents of the first (string) field become the cell labels along one axis.

CHANGING FILE STRUCTURES

You can use the commands `export` and `import` to remove fields from data files completely.

```
import "salegraf_exp" as "profit" logical "p"
```

1 This creates a copy of the exported file `salegraf_exp` as a data file, and opens it in memory.

```
display
```
2 The file has only two fields, `code$` and `profit`

```
close
```
3 Close the file `profit_dbf`.

The common format in detail

The rules for the common format (excluding QUILL) are this:

- Each line in the file ends with a carriage return (ASCII 13 and 10). Lines do not end with commas.
- The end of the file is marked by ASCII 26.
- All other characters are printable ASCII characters (letters, numbers and punctuation).
- The first line must contain names between quotation marks, separated from each other by commas. Each name determines the type of data (text or numeric) that appears in the column beneath it.
- The remaining lines contain as many items (fields) as there are names, with each separated by commas with text strings in quotes.
- Empty strings are represented by `""`, and zero numeric fields by `0`.

Printing elsewhere with SPOOLON

You can use these rules to create a copy of a file with more fields than the original.

```
spoolon"percent_imp"
```
1 `spoolon` diverts all output normally sent to the printer to a file on tape.

```
print"'code$','profit','percent'"
```
2 This determines the fields the file will have: `code$`, `profit`, `percent`.

Note that the whole string appears between double quotation marks, and each field name (with or without the $ sign) appears between single quotes, separated by commas.

Note: you can use single quotation marks as literal characters if they are enclosed by double quotation marks.

```
F5 lprint"'code$','profit','percent'"
```
3 If the line appears correct when printed on the screen, edit the line and put `l` on the front of print.

This sends the line to the spool file on tape.

Note: if you do 'print' a mistake using this method, then `spooloff` and start again.

```
use"s"
```
4 Use the `sales` file and enter the next line:

SPOOLON, SPOOLOFF

Like the `export` command, `spoolon` is designed primarily to create files on tape.

The word has its origins in the days when computers were large and expensive, and many people shared facilities like printers.

In such situations, printing cannot be done whenever anyone feels like it. The print jobs are 'printed' onto tape (spooled) and join a queue of similar files waiting for printing.

`spoolon` tells ARCHIVE that commands like `llist`, `lprint` and `dump` (see next section) should send their output somewhere other than the printer. If the file name is `screen`, it is sent to the screen.

The redirection of printer output continues until the command `spooloff` is used, which closes the 'spooled' file.

The file is an ASCII file like those created by `export`.

Make a file suitable for use by QUILL, by adding `quill` to the end of the command, eg. `spoolon "document" quill`.

`spoolon` creates files with the extension `_lis` unless you specify otherwise.

```
first: while not eof():print"'"+code$+"',";profit;",";0: next:
endwhile
```

5 Be careful when typing this line. If you misplace a comma or mix up double and single quotes, or full and semi-colons, it will not work.

Note that literal quotes are put around the contents of the string field `code$` using `"'"`, while `profit` is merely followed by a comma.

A literal zero is printed to act as the contents of the new field `percent`.

Remember that `print` sends its own carriage return unless suppressed.

The screen now displays:

```
'LLU',420,0
'LLU',380,0
'LLU',1105.5,0
'LLU',871,0
'LLU',1060.5,0
'LLU',240,0
'LLZ',76.5,0
 etc.
```

```
F5 first: while not eof(): lprint"'"+code$+"',";profit;",";0: next:
endwhile
```

6 If the output looks correct, recall the line and add an `l` at the start of `print`.

This time the data is sent to tape.

```
lprint chr(0)+chr(26)
```

7 This command puts an end of file marker (ASCII 26) on the tape.

It is preceded by a null (ASCII 0) because normally the printer driver filters all unprintable ASCII characters from print commands.

You can actually `lprint` a special character or control code by preceding it with `chr(0)`. This tells the driver to let the next character through as itself.

```
spooloff
```

8 This cancels the `spoolon` command, closes the file and saves it on tape.

```
import "percent_imp" as "percent" logical "p"
```

9 This makes a copy of `percent_imp` as an open data file, `percent_dbf`.

```
display
```

10 Display the file. There are now three fields: `code$`, `profit` and `percent`.

```
close
```

11 Close the file.

You now have an assortment of versions of the `toys` and `sales` files on tape 2:

```
sales_dbf                    salegraf_exp
toys_dbf                     profit_dbf
toygraph_exp                 percent_imp
toyfigs_exp                  percent_dbf
salefigs_exp
```

Remember to remove unwanted files while you can still remember whether they are important or not.

You can use this method combined with normal listing techniques to create combinations of fields or files to send to EASEL or ABACUS which `export` could not produce.

QUILL requires special end of line symbols, and obviously no quotes or commas unless literally supposed to appear in the document.

If you have ARCHIVE version 2, you can also use this method with loops and the functions `FIELDN()`, `FIELDT()` and `FIELDV()` to write procedures which add new string or numeric fields to any file structure.

Import errors

There are some fairly common (and at first mystifying) errors you can encounter when importing files into ARCHIVE. Many of them refer to the process of copying in the file, and not to the syntax of the command line itself.

- You cannot import files from QUILL

For more information on `import/export` and integrating the programs, see *QL Integrating the Software*, or the appendix in the QL manual.

Here are some errors, with possible causes:

`Error - String Expected`
One of the field names on the first line is not between quotation marks.

`Error - Wrong Data Type`
A numeric data item (no quotes) appears in a column headed by a field name ending in a `$` sign (or vice versa). Remember that empty strings are `""`, empty numerics are `0`.

`Error - Cannot Open File`
Remember to name the export file to be imported in full, including extension.

`Error - Duplicate Name`
The last command partly succeeded despite an error, and left the new data file open.

Example: Labels and another way of printing files

Nigel has borrowed a few sheets of computer label stationery from the box Joyce acquired for sending catalogues to her customers. This is continuous, sprocket-holed, 'fanfold' paper with two columns of self-adhesive labels across the page.

Nigel sets about preparing a program to use it in preparation for the wedding present thank-you letters and (quite soon, he hopes) change-of-address cards.

He starts by opening and examining the `guests` file, checking it for lines which are too wide for labels.

INSTRUCTIONS

If you have just started, load ARCHIVE, otherwise, save any procedures in memory and use `new` to clear the memory.

The DUMP command

The command dump is another printer command, and lists every record in the file one line at a time.

```
use "g"
```

1 Make guests the current file.

```
spoolon "screen"
```

2 If you do not have a printer, send the output to the screen instead. This is always useful when testing printer output.

```
dump
```

3 After a short pause while ARCHIVE prepares the file for printing, all the records in the file are listed:

```
name$                          address1                        ad
dress2$     address3$          address4$     copa
John and Sue Heath               43b Clarkson Road             Lo
ndon        SW16                           y  2
Sally Jones and Susan Clark      3b Academy Mews               Ch
elsea       London             SW3           y  2
Joshua Grimshaw                  The Old Rectory               Ma
rston Road  Shambles Cross     Cumbria          3
The Paradise Posse               The Athenaeum Theatre·        Br
oad Street  Lyme Stodges       Cumbria          3
Bill Hackney, Drama Critic       The Marston Gazette           22
 High StreetMarston on Tees    Cumbria          1
Jon Ducane                       Northern Lights Dance Company 5
Side        Newcastle on Tyne  Tyne and Wear    1
```

Notice that the entire contents of every field is printed in columns across the screen, with the field name at the top of each column.

The width of each column is set to the length of the longest example of that field in the file. If the resulting column width is less than the field name then the name is abbreviated.

The result is to make the width of each record as wide as the length of all the longest fields put together, which is too wide for the screen. The records have to be *wrapped round* to the next line to fit.

The word 'dump' comes from an old piece of computer jargon. It describes taking the entire contents of a file or of a section of memory and listing it 'as is' on a terminal, whether printer or screen.

Nigel enhances the dump

This is not what Nigel really wanted, as it is almost impossible to use. He improves matters by limiting the fields that dump prints.

```
dump;name$,address1$
```

1 Note that the command is followed by a semi-colon, and the field names are separated by a comma. If you do not type in a



Miranda and Nigel count their blessings

semi-colon after the command, dump will print all the fields as if none had been specified.

This time the printer prints only the fields requested. They are still in columns as wide as the longest field:

```
name$                              address1$
John and Sue Heath                 43b Clarkson Road
Sally Jones and Susan Clark        3b Academy Mews
Joshua Grimshaw                    The Old Rectory
The Paradise Posse                 The Athenaeum Theatre
Bill Hackney, Drama Critic         The Marston Gazette
Jon Ducane                         Northern Lights Dance Company
Dame Edith Throgmorton-Dunne       The Boundaries
Mabel Pew                          12 Acacia Avenue
Philip and Brenda Moore            12 Queens Gardens
Lt-Col and Mrs Anthony Laurence    Overton Grange
Arthur and Janice Blake            131 Jubilee Terrace
Mrs Agnes Blake                    Cherry Cottage
```

The dump command is obviously handy for files whose data can be easily used in a tabular manner, like memos, but offers very little control of how the data will be presented.

```
all: if len(name$)>35:print name$:endall
```

2 Nigel notices some long names and gets ARCHIVE to print them. He can't be bothered to alter them now, but can test his label program anyway.

```
spooloff
```

3 Cancel the spool on screen command

Note: dump prints all the records in the file unless you select them first.

Computer label stationery

Peel-off self-adhesive label stationery for computers usually comes with 2 or more labels across the page. It is practically imposssible to print labels using a printer's friction feed, due to the paper slipping. Use the tractor (sprocket) feed with perforated stationery to keep print and labels in register with each other.

Multi-column labels pose a problem when printing addresses on them. It is easy to print one name and address record at a time, one after the other. It is not so easy to print the names from several records, followed by the first address line from the same records, and so on.

One solution is to store the name and address of each record in variables, then to print the variables using tab to get to the start of each label column.

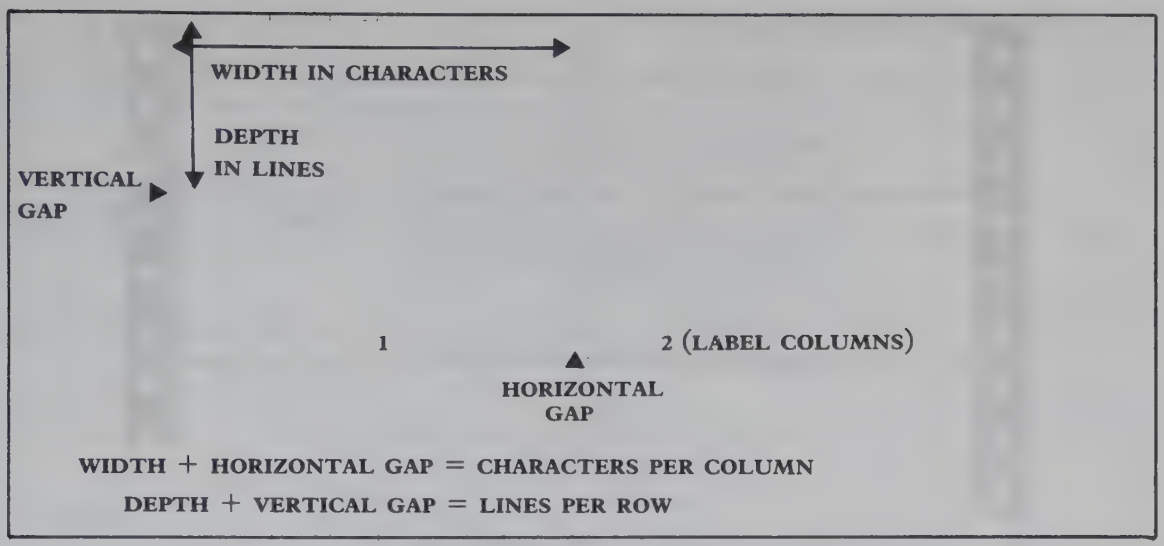


Figure 13.3 – Significant dimensions of label stationery

A program along these lines to print three labels across the page would need the following steps:

```
while not  eof()
 let a1$=address1$
 let a2$=address2$
 let ......
 (etc)

next
 let b1$=address1$
 let b2$=address2$
 let .......
 (etc)

next
 let c1$=address1$
 let c2$=address2$
 let .......
(etc)

next
 lprint a1$;tab 38;b1$;tab 76;c1$
 lprint a2$;tab 38;b2$;tab 76;c2$
 lprint ......
 (etc)

endwhile
```

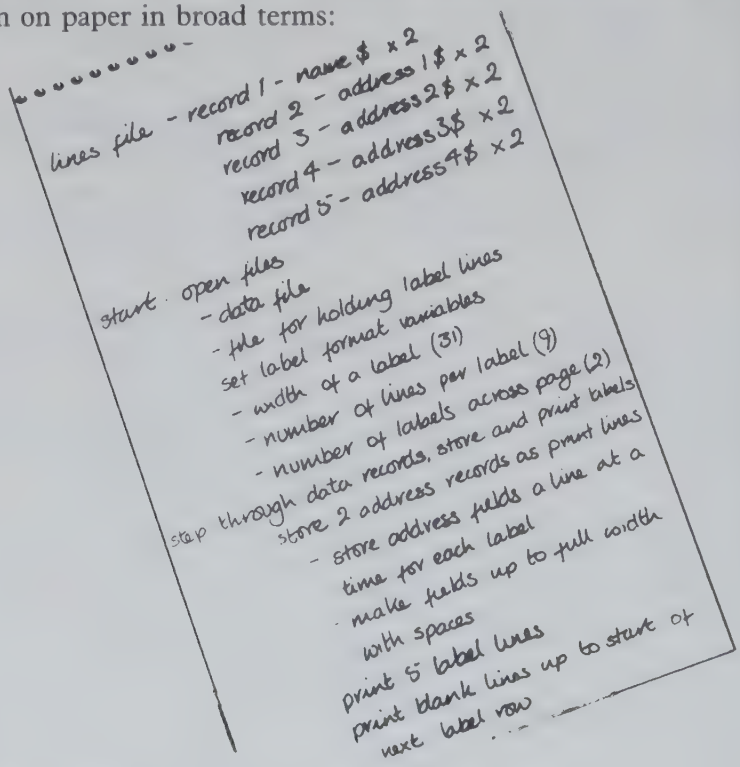
Nigel decides not to adopt this method. Apart from being tedious to type, it is extremely difficult to change. He knows that he may later use different files, or labels with a different shape and a different number of columns across the page.

Using loops to process labels

Nigel can write an adaptable program by using loops to store and print the lines. This means that each line must be able to be identified at some stage by the same variable name.

Nigel decides to use a field called `line$` in a data file to store the label lines. Each record will hold one line as it appears when printed. For example, the first record contains the `name$` fields of two address records.

Before writing any procedures, Nigel outlines the schema of the program on paper in broad terms:



The label line file

Nigel first creates a data file to hold the print lines:

```
create"guests_lbl"logical"lbl" ↵ line$ ↵endcreate
```

1 The file has the same name as the data file with a different extension to differentiate them on tape. This allows the program to be used later with different files, each with its own easily identified `_lbl` file.

```
let x=0:whilex<5:append:letx=x+1:endwhile
```

2 This line appends five completely blank records to the file. The file must contain only as many `line$` records as there are lines to be printed from the address file (`name$`, `address1$`, `address2$`, `address3$`, `address4$`).

```
close
```

3 Close the file to make sure it is properly saved onto tape, and in anticipation of opening it each time labels are required.

Vital statistics

Nigel's first procedure opens the files and sets the label format variables:

```
edit
```

1 Use `EDIT` to create a procedure.

```
proc start ↵ let datafile$="guests"
```

2 Remember that using variables allows programs to be quickly adapted for other files and formats.

```
let df$="g"
```

3 The data file's logical file name will be used to distinguish it from the label line file.

```
rem open datafile$ logical df$
```

4 Insert a line to remind you what the two variables mean and that the data file should be open.

```
open datafile$+"_lbl"logical"lbl"
```

5 Opens the label line file which relates to the data file in use.

Remember that the line file must have exactly as many records as there are fields to be printed from the data file.

Remember that if you use a non-standard extension to create or save a file, you must always use it when reopening the file.

```
let across=2
```

6 The variable `across` determines the number of labels across the page.

```
let width=31
```

7 The width holds the number of characters from the start of one label up to the start of the next one across.

Note that this does not take account of the fact that there is usually a gap of two or three characters between labels. There is obviously no point printing anything in the gap.

Nigel feeds in a temporary value of `31` so that the two labels do not exceed the screen width while he tests the procedures.

```
let lines=9
```

8 The variable `lines` holds the number of lines between the top of a label and the top of the label in the next row down (ie. the height of a label).

This includes the gap of one line between labels.

```
labels endproc
```

9 `labels` is the name of the procedure which starts and governs the rest of the label procedures.

The procedure now looks like this:

```
proc start
  let datafile$="guests"
  let df$="g"
  rem open datafile$ logical df$
  open datafile$+"_lbl"logical"lbl"
  let across=2
  let width=31
```

```
let lines=9
labels
endproc
```

Storing a row

Nigel starts straight in on storing and printing a row, so that he can see some results without finishing the whole program first.

Before printing the label lines which are stored in `guests_lbl`, they must be created from the fields of two different addresses.

```
edit ↵  F3 nlabellines
```

1 Use `EDIT` and start the procedure `labellines`.

```
first "lbl" ↵ while not eof()
```

2 The `first "lbl"` command simultaneously makes `"lbl"` the current file and makes the first `line$` the current record.

There is no need to use a counter, because the `lbl` file only contains as many records as there are lines to be printed. Each line is stored in sequence according to its record number.

```
let fld$=fieldv(recnum("lbl"),df$)
```

3 The function `recnum()` returns the number of the current `line$` record in the `lbl` file, and assigns the value of that field in the data file to `fld$`.

Note that a logical file name can be used as the second argument for `fieldv()`.

For example, with the first `line$`, record 0, `fld$` is assigned `fieldv(0)`, which is the `name$` field in the `guests` file.

Note: early versions of ARCHIVE may produce an error when using logical names with `FIELDV()`. Make the file current first using `use`.

```
let fld$=fld$+rept(" ",width-len(fld$))
```

4 This line adds as many spaces as are needed to make the line as long as a label is wide.

```
labelstore;fld$
```

5 This calls a procedure which stores the field and spaces (passed as a parameter) in `line$`.

```
next endwhile
```

6 This moves onto the next record and the process starts again with the contents of the next data field.

Remember that `first "lbl"` made `lbl` the current file.

The procedure `labellines` should look like this:

```
proc labellines
  first "lbl"
  while not eof()
    let fld$=fieldv(recnum("lbl"),df$)
```

```
    let fld$=fld$+rept(" ",width-len(fld$))
    labelstore;fld$
    next
    endwhile
  endproc
```

Storing a line

```
labelstore;txt$
```

1 Create the procedure `labelstore` which updates `line$` with the `txt$` string passed to the procedure.

```
if n=0 ↵let line$="" ↵endif ↵
```

2 This clears `line$` whenever the first label in a row is about to be stored.

It is done by an `if` clause here because doing it before the loop in `labellines` would mean using several `update` commands to save the blank records.

```
let line$=line$+txt$
```

3 This line adds the passed parameters (in this case, part of the address) to `line$`.

```
update
```

4 The `line$` record is now updated.

The procedure should look like this:

```
proc labelstore;txt$
  if n=0
    let line$=""
    endif
  let line$=line$+txt$
  update
  endproc
```

Printing the row

After storing ten fields from two address records in the five line records, the program uses a loop to print the lines:

```
labelprint ↵first "lbl" ↵while not eof() ↵linepr; line$ ↵next ↵
endwhile ↵
```

1 Write the procedure `labelprint`.

```
linepr;txt$ ↵print txt$ ↵
```

2 Write the procedure `linepr` to actually print the text.

Finally, make up the difference between the five lines printed and the start position of the next label down:

```
labelnewrow ↵let count=0 ↵while count<lines-recnum() ↵
```

3 Start a new procedure `labelnewrow`
It uses a counter loop to print as many carriage returns as the total depth of the label less the number of lines already printed.

This number is the same as the current `_lbl` file record number, which represents the last line printed.

Note: Remember that record (and field) numbers start counting from 0 as the first entry.

```
linepr;"" ↵let count=count+1 ↵endwhile ↵
```

4 Finish the procedure by using `linepr` with no text to print a carriage return, then increment the counter and end the loop.

The procedure should look like this:

```
proc labelnewrow
  let count=0
  while count<lines-recnum()
   linepr;""
   let count=count+1
   endwhile
  endproc
```

```
ESC ESC save"labels1" ↵save"mdv1_labels1" ↵
```

5 Save the first generation copy of the program onto both tapes.

Make sure you have a backup cartridge in drive 1, and save the procedures in case some accident destroys them before you get a chance to use them.

Use two command lines rather than a colon to separate the two `save` commands, so that if either command fails for any reason, you know which one.

```
start
```

6 Set the format variables and open the `labels` file by entering `start`.

Ignore the error message, as `labels` has not yet been written.

```
labellines:labelprint
```

7 Type these two commands to try it out.
After a pause while the program stores two addresses, two labels appear side-by-side on the screen. As the `line$` fields are `updated`, the data drive may at first whirr intermittently. Once the lines remain constantly the same length, this should stop.

The labels appear like this:

```
The Paradise Posse              Jon Ducane
The Athanaeum Theatre           Northern Lights Dance Company
Broad St                        5 Side
Lyme Stoges                     Newcastle on Tyne
Cumbria                         Tyne and Wear
```

The controlling procedure

Nigel now writes the main, controlling procedure which steps through all the records and from which all others are called. The procedure is named `labels`, and is called initially from `start`:

`labels ↵ cls ↵ while not eof(df$)`

1 Use the logical name of the data file (stored in `df$`) for clarity and to ensure the correct file is tested by `eof()`.

`let n=0 ↵ while not eof(df$) and n<across`

2 The variable `n` is the counter for the number of labels in a row. The loop fails when two passes have been made (when `n` is equal to `across`).

Note that `eof()` is used again in this loop. Failure to do so causes the program to stay in this loop until the row is finished, even if the file has ended.

This means that if the last record is the first label in a row, it will be repeated until the row ends.

`labellines` 3 This calls the procedure described above which stores one address as part of the label lines.

`next df$` 4 The event which occurs most often (apart from storing part of a line) is stepping through the address records. `next` thus appears in the deepest loop.

Common mistakes in `eof()` procedures include repeating the `next` after the deepest loop, or leaving it out altogether.

The logical file name is used for clarity and to make `guests` the current file.

`let n=n+1` 5 Increment the loop counter
Although there is an `eof()` in the `while` condition, `n` (which is also in the `while` expression) causes the loop to fail every two records, `next df$` only once per file.

`labelprint` 6 Calls the procedure which prints the stored data lines across each label in a row.

`labelnewrow ↵ endwhile`

7 Calls the procedure which prints carriage returns from the last data line down to the top of the next label.

`close "lbl"` 8 Close the label lines file as soon as the program finishes.

The procedure now looks like this:

```
proc label
 cls
 while not eof(df$)
   while not eof(df$) and n<across
     labellines
     next df$
     let n=n+1
```

```
    endwhile
  labelprint
  labelnewrow
  endwhile
close "lbl"
endproc
```

ESC ESC save"labels2" ↵ save"mdv1_labels2"

9 Save the second generation version of the program.

Notice that the program never uses a form feed (ASCII 12) to eject the page. Label stationery usually has exactly the same gap between pages as between two rows of labels. The length of the page is irrelevant. Once lined up correctly, paging takes care of itself.

Test print

To help line up the labels correctly before proceeding with printing, offer the option to print a row of test labels. This prints asterisks where the address lines start. If they are out of line with the labels, adjust the platen (the roller) or tractor/sprocket feed positions and do another test.

edit 1 Use EDIT to create the procedure labellineup.

```
proc labellineup
 let ok=0
 while not ok
   print at 13,10;"Test print ? (Y/N)";
   let testprint$=upper(getkey())
   print testprint$
   if testprint$="N"
     let ok=1
     else
     if testprint$="Y"
     let n=0
     while n<across
       labellines
       let n=n+1
       labelprint
       labelnewrow
       endif
     endif
   endwhile
 endproc
```

F4 if testprint="N" ↵ ↵ ↓ else ↵ let fld$="*" ↵ endif ↵

2 TAB to the procedure labellines and add these lines so that an asterisk is stored instead of a field when testprint$ is Y.

The `if` clause storing `fld$` now looks like this:

```
if testprint="N"
 let fld$=fieldv(recnum("lbl"),df$)
 else
 let fld$="*"
 endif
```

`labellineup` 3 TAB to the procedure `start` and insert the call `labellineup` just before `labels`.

`progmenu` 4 Add the line `progmenu` immediately after `labels`. This is explained in the next section.

ESC ESC `first "g"` ↵ `start`

5 Move back to the first `guests` record and try out the test print facility. Ignore the error message about `progmenu` when the program finishes. It has not been written yet.

The test print looks like this when printed on the screen:

```
                Test Print ? (Y/N) Y
*                                  *
*                                  *
*                                  *
*                                  *
*                                  *


                Test Print ? (Y/N) N
```

The screen then clears and the following is printed:

```
The Paradise Posse          Jon Ducane
The Athanaeum Theatre       Northern Lights Dance Company
9 Broad St                  5 Side
Lyme Stoges                 Newcastle on Tyne
Cumbria                     Tyne and Wear
```

etc.

`save "labels3"` ↵ `save "mdv1_labels3"`

5 Save the program as `labels3`.

A program menu

Nigel has named all the procedures with the same first letters so that they stay together when merged with a longer program.

However, he realises that the program opens another file, and might run into memory difficulties if used with large programs, a screen layout and other open data files, especially if ordered. The solution is to `run` rather than `merge` programs. Merging programs increases the number of procedures in memory, running them replaces one set with another.

He writes a menu to offer a choice of different programs which can be `run` (loaded and automatically started). For the wedding files these include `wedcom`, `lists`, `labels` (for another pass) and `wedfiles` (for backing up files).

`load "dummy"` 1 Load the empty program file to clear the label procedures from memory.

The next procedure should be saved alone for easy duplication.

`edit` ↵ F5 `progmenu` ↵

2 Rename the procedure `dummy` to `progmenu`.

```
cls
print at 0,20; "WEDDING PROGRAMS"
print at 5,5; "1. Maintain Guest List"
print at 6,5; "2. List and Print Guests"
print at 7,5; "3. Print Labels"
print at 8,5; "4. Backup Files"
print at 9,5; "Q. Quit ARCHIVE"
print at 12,5; "Press letter of chosen option";
while 1
  let c$=upper(getkey())
  print c$
  if c$="1":run "wedcom":endif
  if c$="2":run "lists":endif
  if c$="3":start:endif
  if c$="4":run "wedfiles":endif
  if c$="Q":quit:endif
  endwhile
endproc
```

3 Type in the rest of the procedure and

ESC ESC `save "progmenu"` ↵ `save "mdv1_progmenu"`

4 Save the procedure.

`load "labels3"` ↵ `merge "progmenu"`

5 Reload the labels program and merge the program `progmenu`.

6 Create a remarks file, `a:`

```
proc a
  REM               labels_prg     10 5 84 12.00
  REM start: opens data and array file, sets label values
  REM labellineup: prints asterisks at label start for a row
  REM labels: main file and row loops; stores,prints
  REM labellines: prepare data fields for label lines
  REM labelstore: passed field to add to line$
  REM labelprint: prints label lines (array records)
  REM labelnewrow: new lines to next row
  REM linepr: passed text to print or lprint as line
  endproc
```

```
save"labels" ↵ save"mdv1_labels"
```

7 Save the final version of the program.

```
kill"labels1_prg" ↵ kill"labels2_prg" ↵ kill"labels3_prg"
```

8 Remove the development versions from the tape. Use F5 to repeat each line, the CTRL and arrow keys to jump to the number, and change the number to save retyping the line in full each time.

How to use the program menu

To make the program menu work with the other programs, the procedure `progmenu` must be merged with each of them, and then called when they are finished (eg. when `c$` is `S` for `stop` in `wedcom`).

It is needed because running a file wipes out every current procedure, even the one that used the command.

The programs `lists` and `wedfiles` will have to be tidied up and given menu and start procedures before being any real use with this system. Remember also that `run` without a `start` procedure causes an unrecognised command error.

This method sacrifices speed for space and ease.

Arrays and string handling

If you are familiar with programming languages like BASIC, you probably noticed that Nigel is using the `lbl` file as a one-dimensional array in his label program.

There is no need to understand exactly what an *array* is to benefit from using one of their major advantages: the ability to designate variables by the number of their position in a list of variables. In practical terms, this means that you can use counters to vary the variables used in loops.

Arrays

The following diagram illustrates a 2-dimensional array (also known as a matrix or a table) of data.

```
   1          2                3          4
  CODE       TOY              COST       PRICE
1 LLU      Luny Llama         3.45        8.5
2 LLZ      Lawrence Lizard    2.34        9.99
3 PUD      Pudgey Penguin     1.7d        5.5
```

ARCHIVE data files are themselves 2-dimensional arrays (2 axis with 2 co-ordinates per item). Any item can be uniquely specified by name (eg. cost of lizard) or number (eg. third field, second record).

ARCHIVE files have high overheads in terms of memory, however, and occasionally use the tape drives, slowing some operations unnacceptably. It can be preferable to create arrays from ordinary variables.

A 'single-dimensional array' is simply a fancy way of describing a list of linked items.

In Nigel's label program there are several variables called `line$` in existence at once – `line$` record 1, `line$` record 2, `line$` record 3 and so on. Each is called `line$`, so a single command can affect each of them. Different `line$`s can be brought into play by using a number (the record number) to specify which one.

The alternative is creating a fixed number of differently named variables and writing one command per line, eg: `print line1$ print line2$ print line3$` etc.

This wastes a lot of space in memory, is tedious to type, hard to adapt and inherently inflexible.

ARCHIVE was originally conceived as a database with powerful language facilities, rather than a language with powerful file handling facilities (which it also is). It does not at the moment include array handling in its repertoire.

Arrays do, however, have some useful properties, which can be emulated in a variety of ways.

Strings as fixed length arrays

Another method of imitating the benefits of a one-dimensional array uses strings to store the variables.

The following procedure extracts fixed length substrings from a larger string according to their position (number) in the string:

```
proc listmonths
  let m=0
  let m$="JanFebMarAprMayJunJulAugSepOctNovDec"
  while m<12
    let start=(3xm)+1
    print m+1;" ";m$(start to start+2)
    let m=m+1
    endwhile
  endproc
```

This displays `1 Jan 2 Feb 3 Mar` and so on.

Each month is a fixed substring 3 letters long, and this technique can be used with any fixed length as long as the literal numbers (3, 2 and 12 above) are changed to suit.

There is a limit of 255 characters (the maximum length of a string) when using this technique on the combined lengths of the substrings. This makes this option of no use for labels (eg. 5 lines × 3 columns × 35 characters = 525 characters).

For illustration purposes this example uses a string literal provided at the start. In practice, it could be created and maintained elsewhere and contain changing data.

So, to change the 7th substring in the above example:

```
let start=(3x6) + 1
let m$=m$(to start-1) + newsubstring$ + m$(start+3
to)
```

The new substring must, of course, be exactly 3 characters long.

Slicing variable length substrings

Variable length substrings can be stored in and extracted from a string 'array' without the need to set a fixed length equivalent to the longest:

```
proc listblakes
  let n$="Harold*Joyce*Miranda*Chris*Paula*"
  let laststar=0
  while laststar<len(n$)
    let nextstar=laststar+instr(n$(laststar+1 to),"*")
    print n$(laststar+1 to nextstar-1);" Blake"
    let laststar=nextstar
    endwhile
  endproc
```

This displays:

```
Harold Blake
Joyce Blake
Miranda Blake
Chris Blake
Paula Blake
```

This procedure searches for a *delimiter* (in this case an asterisk) which marks the start and end of substrings.

Remember: `INSTR()` returns not simply 1 if a substring is found in a string, but the number of its starting position.

Each search for the next star begins after the last star, and the name is the substring between the last star and the next star.

The loop stops when `nextstar` is the same as the position of the final star in the string. Obviously, the character used as a delimiter must never appear as part of a substring itself.

A dry run of the procedure looks like this:

```
Times through loop      1       2       3       4       5

let laststar=           0
while laststar<len(n$)
  let nextstar=         0+7=    7+6=    13+8=   21+6=   27+6=
                        7       13      21      27      33
  print n$ slice        1-6     8-12    14-20   22-26   28-32
  let laststar=         7       13      21      27      33
endwhile
```

Numeric values can be stored in the string by first converting them using `STR()`. This can be used to effectively increment a loop counter in irregular steps. For example, store a string ' 0 4 5 ' and extract the values in turn to use with `FIELDV()` in a loop. This then only affects the first, fifth and sixth fields.

Slicing address lines

The sequential slicing procedure can be useful with address records. Many mailing lists need to be ordered at different times on a variety of fields, such as individual name, company name, town, county or country. ARCHIVE can only `order` a file on a specific field, so each item will need one of its own. However, there may be from one to three lines of house, street and area address, and if three fields are allowed for, some will often be blank. This causes problems when printing labels.

One solution is to store all the other address lines as one field, delimited by commas, and then slice the line accordingly:

```
company$:  Old Town Crafts Ltd
address$:  Unit 6, First Avenue, Waterford Industrial Estate,
town$   :  Kingshill
county$ :  Ayrshire
```

```
proc sliceaddress
  let delim$=","
  let last=0
  while last<len(l$)
    let next=last+instr(l$(last+1 to), delim$)
    print l$(last+1 to next-1)
    let last=next
    endwhile
  endproc
```

Note that this procedure uses the commas solely as delimiters, and does not print them.

More about blank address fields

An alternative approach to the problem of blank address lines uses a counter in `labellines` to count the fields, and only stores lines with something in them. Another loop then stores blanks to as many of the `line$` fields as were not filled with address fields, eg.

```
proc labelines
  first "lbl"
  while count<numfld("lbl")
    let fld$=fieldv(recnum(count),df$)
    if fld$>" "
      labelline;fld$+rept(" ",width-len(fld$))
      next
      endif
    let count=count+1
    next
    endwhile
  while not eof()
    labelline;rept(" ",width)
    next
    endwhile
  endproc
```

Folding long lines

The slicing technique can be used to wrap or fold long lines of words or addresses to fit within a maximum line length.

The following procedure begins by looking at the maximum position and works backwards until it finds a space. The space is being used here as a delimiter to identify the start and end of words. The line is cut at the first space, then the rest of the line is repeatedly searched while it remains too long.

The procedure is passed two parameters when it is called – the long string to be folded (`line$`) and the maximum length of a line (`max`):

```
proc fold;line$,max
  let start=1
  let pos=max
  let length=len(line$)
  while pos<length
    while line$(pos)<>" "
      let pos=pos-1
      endwhile
    print line$(start to pos-1)
    let start=pos+1
    let pos=pos+max
    endwhile
  sliceline;line$
  endproc
```

Notice that there are no `if` clauses in the procedure. The conditions governing the execution of the loops fail automatically when there is no more work to do.

Nigel could use a mixture of this procedure and the previous example (with a few modifications) to fold overlong name lines in his label program.

Numbered delimiters

This particular method is best suited to extracting substrings in sequence.

They can be found by number instead if a number is stored after the first delimiter of the substring, eg:

```
proc listblakes
  let n$="*1Harold*2Joyce*3Miranda*4Chris*5Paula*6"
  let max=val(n$(len(n$)))
  let count=1
  while count<max
```

```
    let findnum$="*"+str(count,2,0)
    let laststar=instr(n$,findnum$)
    let nextstar=laststar+instr(n$(laststar+1 to),"*")
    print n$(laststar+2 to nextstar-1);" Blake"
    let count=count+1
    endwhile
endproc
```

Notice that the starting position of the printed substring has been increased to cut out the number, and that a number has been stored at the end of the string to provide a maximum number of times around the loop.

There is a limit to the length of the major string, but several can be stored as records in a file. The slicing loop can then be nested inside another loop which steps through the records.

This is virtually a case for using fields instead of substrings, but the mixture of substrings and files can be useful.

For example, it could form the basis of a 3-dimensional array with the three axes: records, fields and substrings.

Multi-dimensional arrays can be used for plotting graphs, moving graphics about the screen, multi-choice single-screen menus (all using `print at`) or multiple levels of subtotalling.

In practise, records and fields can often be used instead of strings as arrays for many purposes. Decide which to use according to the requirements of the task in hand.

These methods of storing and using variable items are slower than using one variable per item, but the procedures are very compact and flexible.

The function VALUE()

In ARCHIVE version 2, the function `VALUE()` can be used to extract data from arrays of variables.

`VALUE ()` returns the contents of the variable named by its argument.

For example, `print value ("name$")` might display `"John"`.

This allows information to be stored in a series of variables (var1$, var2$, var3$ etc.) and then be extracted by a `count` loop with a line like this:

```
print value ("var"+str(count,3,0)+"$").
```

If `count` were 2, ARCHIVE would read the line as:

```
print value ("var2$").
```

Procedures which write procedures

Commands and field names themselves cannot be stored in variables and then used by using the variable instead. For example, you cannot ask for a `selection` of the `guests` file with these commands:

```
let choice$="address3$":select choice$="London"
```

The command `order` is another example. A file can contain many fields, but writing a procedure to offer a choice of sequences entails using the command over and over again, eg:

```
if choice$="1":order name$;a:endif
if choice$="2":order company$;a:endif
if choice$="3":order town$;a:endif
(etc)
```

There is a way of getting around this and using a single procedure to order according to choice.

First create an ASCII file on tape (using `spoolon`) which looks just like a `_prg` file, and then `merge` the result.

This is slow but inherently flexible. The following program offers ordering on any field of any file.

The controlling procedure `reorder` calls four procedures which execute each step in turn:

```
proc reorder
  cls
  listfields
  choosefield
  writeproc
  screen
  endproc
```

```
proc choosefield
  let ok=0
  while not ok
    input at 13,10; "Order on which field ? "; num
    if num>=0 and num<=numfld()
      let ok=1
      endif
    endwhile
  endproc
```

```
proc listfields
  let count=0
  while count<numfld()
    print tab 10; count; ". "; fieldn(count)
    let count=count+1
    endwhile
  endproc
```

```
proc writeproc
  spoolon "sequence_prg"
  lprint "proc putinorder"
  lprint "order "+fieldn(num)+";a"
  lprint "endproc"
  lprint chr(0)+chr(26)
  spooloff
  merge "sequence"
  putinorder
  endproc
```

If the `guests` file is used and the number 5 pressed, this produces the following procedure:

```
proc putinorder
  order send$; a
  endproc
```

Complex and varying selections

A more suitable candidate for this treatment is `select` or `search`. Many listing and reporting programs need to offer a complex range of selections – ranges, `INSTR()` searches, exact matches, either/or options etc. The more fields a file has, the more complex this can be.

The following example illustrates some steps that could be taken to write a variable selection program:

Get the choices:

```
print "CHOOSE SELECTION CRITERIA (Press ENTER for all)"
input "Which code ?";selcode$
input "Which month ?";selmonth
```

Store the expression:

```
let selif$=""
if selcode>""
   let selif$=selif$+" and code$='"+selcode$+"'"
   endif
if selmonth>0
   let selif$=selif$+" and month="+STR(selmonth,2,6)
   endif
```

Write the procedure:

```
if selif$>""
  spoolon "selection_prg"
  lprint "proc selectif"
  lprint "select 1=1"+selif$
  lprint "endproc"+chr(0)+chr(26)
  spooloff
  endif
```

Merge and use the procedure:

```
merge "selection" ↵ selectif
```

If `LLZ` and 6 are chosen, the procedure produced looks like this:

```
proc selectif
  select 1=1 and code$='LLZ' and month=6
  endproc
```

Note that `select 1=1` is a trick to enable any following condition to begin with `and`. One always equals one and evaluates as true, so `select 1=1` is the same as selecting everything (ie, no selection at all), letting the other conditions govern the command.

Object files

Procedures can be written like this because they are stored as ASCII files. Each time a procedure is run, ARCHIVE reads each command letter by letter, and converts it into instructions the QL can understand.

It is possible with version 2 of ARCHIVE to save and load procedures with their commands converted to a single code or 'token'. This takes up much less space on tape, and so loads much faster. These tokens are also known as 'object' code.

The command `object` saves procedures in tokenised form, for example: `save "wedcom" object`.

Program files saved with `object` are given the file extension `_pro` rather than `_prg`, and can be opened by putting the word `object` after `load`, `merge` or `run`, eg: `run "wedcom" object`.

It is still possible to `edit` procedures which have been saved this way, but only with ARCHIVE's text editor.

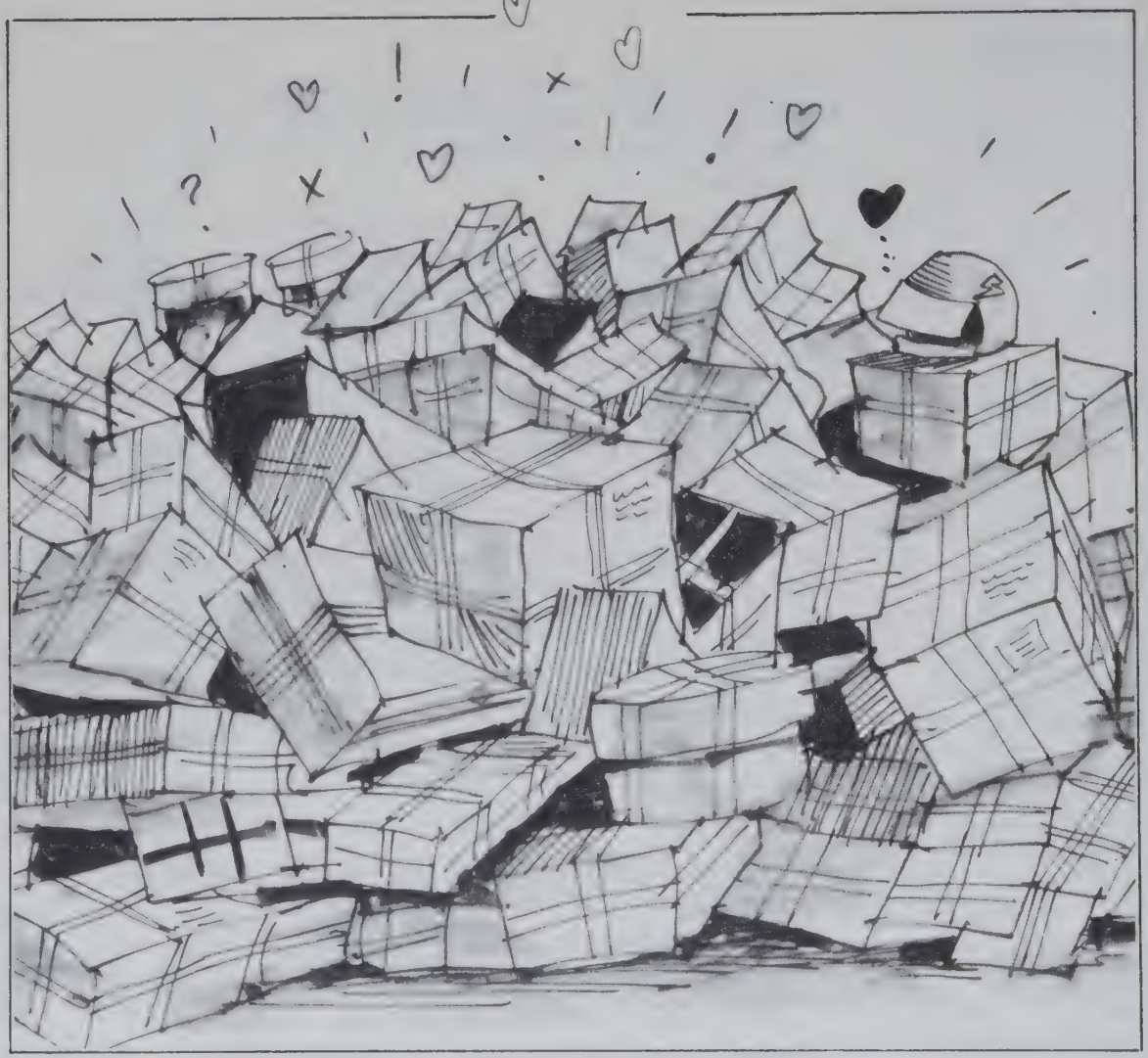
You can even prevent this if you want, by using the word `protect` with `save`, eg: `save "wedcom" protect`.

When loaded again using `object` the protected procedures cannot even be looked at with `edit`.

Helpful hints

There are several ways of helping ensure that programs are written so that they are small and can be simply adapted:

- Use global variables rather than literals for screen or print formats, file names and conditions in loops. Ask for or set these at the start of the program.
- Use variables or files as arrays handled by loops.
- Keep procedures as small as possible and limited to a single purpose. Put lines related to other functions in another (called) procedure, or leave them in a calling procedure if that is where they really belong.
- Use parameters to ensure that the correct type and number of variables are available for any particular task.
- Use the same variable name for common purposes, like flags, output and counters. Make them `local` to the procedure they are used in.



Alone at last

Listing of label printing program

The label printing procedures should look like this:

```
proc a
 REM                labels_prg    10 5 84 12.00
 REM start: opens data and array file, sets label values
 REM labellineup: prints asterisks at label start for a row
 REM labels: main file and row loops; stores,prints
 REM labellines: prepare data fields for label lines
 REM labelstore: passed field to add to line$
 REM labelprint: prints label lines (array records)
 REM labelnewrow: new lines to next row
```

```
 REM linepr: passed text to print or lprint as line
 endproc
proc labellines
  first "lbl"
  while not eof()
    if testprint$="N"
      let fld$=fieldv(recnum("lbl"),df$)
      else
      let fld$="*"
      endif
    let fld$=fld$+rept(" ",width-len(fld$))
    labelstore; fld$
    next
    endwhile
  endproc
proc labellineup
  let ok=0
  while not ok
    print at 13,10; "Test Print ? (Y/N) ";
    let testprint$=upper(getkey())
    print testprint$
    if testprint$="N"
      let ok=1
      else
      if testprint$="Y"
        let n=0
        while n<across
          labellines
          let n=n+1
          endwhile
        labelprint
        labelnewrow
        endif
      endif
    endwhile
  endproc
proc labelnewrow
  let count=0
  while count<lines-recnum()
    linepr;""
    let count=count+1
    endwhile
  endproc
proc labelprint
  first "lbl"
  while not eof()
    linepr; line$
    next
    endwhile
  endproc
proc labels
  cls
  while not eof(df$)
    let n=0
    while not eof(df$) and n<across
```

```
      labellines
      next df$
      let n=n+1
      endwhile
    labelprint
    labelnewrow
    endwhile
  close "lbl"
  endproc
proc labelstore; txt$
  if n=0
    let line$=""
    endif
  let line$=line$+txt$
  update
  endproc
proc linepr; txt$
  print txt$
  endproc
proc start
  let datafile$="guests"
  let df$="g"
  REM open datafile$logical df$
  open datafile$+"_lbl"logical "lbl"
  let across=2
  let width=30
  let lines=9
  labellineup
  labels
  endproc
```

APPENDIX A

Getting out of trouble

GENERAL POINTS

Remember the **HELP** key, **F1**. This is available at any time and does not lose your place.

If the computer stops responding to keystrokes

- Press the **ESCAPE** key or the **ENTER** key if in trouble, they return you to familiar territory.
- If the computer continues not to respond, you may need to reset the machine. You will lose any material in current memory which you have not saved onto a Microdrive cartridge. To reset, remove the cartridge and press the button on the right hand side of the QL. This brings you to the entry screen, where you can call up ARCHIVE by reinserting the program cartridge and pressing **F1** or **F2**.

If the computer does not appear to load ARCHIVE

- Is the screen connected to the computer?
- Are the screen and the QL connected to a power source?
- Do you have an ARCHIVE program cartridge in the left-hand Microdrive?

If the answer to all the above is *Yes*, reset the machine.

If a Microdrive runs without stopping

This happens when you ask for a directory of an unformatted tape. Remove the cartridge, if any,from the drive which is *not* running, and press the reset button. Format the unformatted cartridge.

File names

- Do not use blank spaces in names. Also, do not use '_' at the start of a file name. ARCHIVE does not check this type of filename for correctness. You could crash ARCHIVE by, for example, inserting a space after '_'. The '_' character is used to gain access to a networked system or other hardware attached to your QL.
- A file name can be no longer than 8 characters and should contain only letters of the alphabet or numbers. It should start with a letter of the alphabet.

Long loading time

- Microdrives become very inefficient as they become full. As the tapes start approaching 90% of capacity, there are fewer and smaller slots available for saving files. These can become very fragmented and the cartridge may need to be read many times before the complete file is loaded.
- As soon as your files appear to be taking a long time to load, use the **BACKUP** command to copy each file on to a fresh formatted cartridge. Once all the files have been copied successfully, erase your original files. You can now re-use this clean cartridge. The copying process brings all the separate parts of the file into one contiguous area of the tape, and hence cuts down the number of times the QL has to access the Microdrive.

Difficulty loading or copying tapes

- If you cannot load any of the four software packages supplied with the QL, try making a copy of the cartridge from `mdv2` to `mdv1`. Making a copy this way often produces a cartridge that will run successfully on `mdv1`.

  You can do this by placing the cartridge to be copied in `mdv2` a cartridge in `mdv1`, then type: `lrun mdv2_clone` (See Chapter 1 – First Steps).

- If `CLONE` fails to make a complete copy, try `CLONE` again with a different cartridge in `mdv1`, or try copying the file it failed to copy, using the SUPERBASIC command **COPY**, or the ARCHIVE command **BACKUP**:

  ```
  copy mdv2_filename to mdv1_filename
  ```

  If you succeed in copying the file, copy the remaining files from `mdv2` to `mdv1` (if you want to create a blank cartridge to receive files, use `format mdv1`).

Corrupted files

- Files saved on tape are sometimes impossible to load or open again. Messages like `error reading file`, `incomplete file transfer`, `wrong file type`, and `out of memory` can all be caused by corrupted files, ie. garbled or damaged tape files.

Possible causes include:

- Switching the QL on and off with tapes in the drives
- Resetting the QL with tapes in the drives
- Resetting the QL while in `INSERT`
- Power surges on the QL or Microdrives
- Physical damage to the cartridge or tape (kinking, grease and dust, excessive wear)
- Saving files when the Microdrives are too hot
- Exposing cartridges to strong magnetic fields
- Exposure to static charges (nylon carpets, TV screens etc.)

The only sensible way of avoiding these problems is to make constant and multiple backup copies of your files.

Rescuing a corrupted Data File

If you have no backup copies of a data file which becomes corrupted, you just might be able to save the good records.

Try opening the file with `LOOK` and `DISPLAY` it. If this works, search for the first corrupted record with this command:

```
all:sprint:print recnum():endall
```

Now find the last corrupted record like this:

```
last:while recnum()>0:back:sprint:print rec-
num():endall
```

Use the transfer method described in Chapter 5 to create an empty copy of the file structure, then append all the records up to and after the bad patch in the corrupted data file.

You will have to recreate the lost records from printouts or the original source material.

ERROR MESSAGES

SUPERBASIC error messages

There are some error messages which come from SUPERBASIC, and usually appear while loading ARCHIVE, or making a copy or the program cartridge using `clone`. (Programs called by `lrun` such as `boot` and `clone` are BASIC programs.)

These are messages such as:

```
at line 200 bad lineat line 200 bad medium
```

These messages usually mean that the program is corrupted or the tape damaged, and needs to be re-copied (using `clone`) from the master ARCHIVE cartridge.

ARCHIVE Error Messages

There are two types of error message which can appear when using ARCHIVE, those with an error number, and those without.

Unnumbered error messages

`Bad Line`
ARCHIVE makes certain checks on lines being typed in while editing procedures. If it finds anything unacceptable, it aborts the line completely, and prints this message in the WORK AREA.

eg `print to$`

because `to` is a reserved word, being part of the string slice.

`No files on Microdrive`
or
`No files found on selected Microdrive`

This message does *not* appear if there really are no files on a formatted drive - `dir` always shows the volume name and sectors at least. If the tape is unformatted, the drive whirrs without stopping.

The message usually means that the wrong drive name has been used, eg. `mdv1` without an underline, or `mdv3` where there is no third Microdrive. It also appears when there is no tape in the drive, or it is not pushed all the way in.

This message sometimes appears erroneously if you use `dir` after an error message like `Incomplete file transfer` (error 91) or when you change cartridges in a drive. You will probably have to reset the computer.

`No help available`This message means that the ARCHIVE program cartridge is not in drive 1, or the program cartridge does not have the file `archive_hob` on it. It does not mean that the Help filehas no information on a particular command or situation.

Numbered error messages

When ARCHIVE detects an error in a command typed from the command line or in a procedure, it displays an error number and an error message, in the format:

```
Error 1: command not recognized
```

This error number is returned by the function `ERRNUM()`.

1 `Error: command not recognised`
Cause: usually misspelling a command or calling a procedure not currently in memory.

eg `apend`
should be: `append`

2 `Error: end of statement expected`
Cause: ARCHIVE finds more information on a command line than it expects.

eg `let x=3 let y=4`
should be: `let x=3:let y=4`

It also occurs when you fail to pass a parameter to a procedure which expects one. This is because ARCHIVE does not expect the `proc` line of the called procedure to have any parameters.

3 `Error: variable name expected`
Cause: ARCHIVE expects the next input to be a variable and does not find one.

eg `create "file"` ↵ ↵
should be: `create "file"` ↵ `field$` ↵ etc.

because ARCHIVE expects at least one variable name to be set up in a file. Attempting to use a key word as a variable name in `CREATE` causes the same error.

4 `Error: unrecognized print item`
Cause: after a `PRINT` command, ARCHIVE expects either a variable name, an expression, or characters within quotes. A key word cannot be a print item.

eg `print create`

5 `Error: wrong data type`
Cause: passing a string parameter to a procedure expecting a numeric one, or vice versa.

6 `Error: numeric expression expected`
Cause: trying to assign text to a numeric variable

eg `let x="yes"`
should be: `let x=25`

7 `Error: string expression expected`
Cause: trying to assign numbers to a string variable, or using a numeric variable as a file name.

eg `let x$=25`
should be: `let x$="yes"`

eg `open x`
should be: `open x$`

8 `Error: variable not found`
Cause: referring to a variable which does not exist. This is often because keywords are misspelt, causing ARCHIVE to assume the word must be a variable.

eg `while not eof`
should be: `while not eof()`

eg `print tab61;note$`
should be: `print tab 61;note$`

9 `Error: variable undefined`
Cause: trying to make a command act upon a variable which had not been previously defined or declared. You are unlikely to see this message, as undefined variables usually produce the message above (error no. 8).

eg `print z` (z undefined)

10 `Error: missing separator`
Cause: two expressions on the same line which should be divided by a separator (semi-colons and commas). Usually caused by missing commas in lists, (eg. parameter or input), or semi-colons in print lists.

eg `print at 5;note$`
should be: `print at 5,1;note$`

11 `Error: name too long`
Cause: assigning a name of more than 13 characters to a variable

eg `let zzzzzzzzzzzzzzzzzzzzz=13`
should be: `let zzzzzzzzzzzz=13`

12 `Error: duplicate name`
Cause: opening a file with a logical name that already exists, or assigning a variable name twice.

eg `open "memos" logical "m" open "monthly sales" logical "m"`
should be: `open "memos" logical "m" open "monthly sales" logical "ms"`

eg `create:address$:address$:endcreate`
should be: `create:address1$:address2$:endcreate`

13 `Error: string literal expected`
Cause: Trying to `IMPORT` a file that is not in the format that ARCHIVE expects. For example, ARCHIVE expects the first information in imported files to be the field names of the new data file. These must be in quotes.

14 `Error: missing endproc`
Cause: should only occur with program files constructed using a text editor other than ARCHIVE's own, because ARCHIVE's text editor automatically supplies the `endproc` statement

15 `Error: bad proc statement`
Cause: see error 14

16 `Error: premature end of statement`
Cause: ARCHIVE expects a further expression or command

eg `create"memos":endcreate`
should be: `create"memos":note$:endcreate`

17 `Error: program structure fault`
Cause: usually means a missing endall, endif or endwhile in a procedure. Can also be caused by adding a superfluous endproc.

18 `Error: too many numbers`
Cause: indicates that you are trying to input more numbers than will fit into the memory reserved for input. The error may occur either in a line of input from the keyboard, or while loading a program that includes a procedure with many numbers in one of its lines. The exact limit depends on circumstances – a typical limit would be 15 to 20 numbers, so you are unlikely to get this error message.

21 `Error: file already exists`
Cause: Version 2 only. Commands like `BACKUP` and `CREATE` are not allowed to wipe out an existing file with the same name.

27 *Cause:* When the **ESCAPE** key is pressed while a procedure is running, the function `ERRNUM()` returns the value `27`. **ESCAPE** causes a procedure to stop exactly as if an error has been encountered. 27 is the ASCII code for the **ESCAPE** key.

50 `Error: missing closing quote`
Cause: not necessarily missing a closing quote, but having an odd number of quotes in a command line. ARCHIVE expects quotes to come in pairs on either side of string literals.

eg `open "memos" logical m"`
should be: `open "memos" logical "m"`

51 `Error: missing exponent after E`
Cause: `E` needs to be followed by an exponent

eg `x=1.2E`
should be: `x=1.2E5`
because the scientific representation of numbers requires that the value before the `E` must be raised to a power of 10 represented by the value after the `E`. In this case `x` becomes 1200000.

52 `Error: number too big`
Cause: ARCHIVE can only handle numbers whose value falls in the range 1E-38 to 1E+38.

eg
`print 10000000000000000000000000000000000000`

53 `Error: unknown symbol`
Cause: putting a symbol in a position in an expression where ARCHIVE expects a number or character

eg `let x=%`

70 `Error: evaluator syntax-errror`
Cause: the expression to be evaluated (added, subtracted etc.) is incomplete or incorrect

eg `let x=3+`

71 `Error: mismatched parentheses`
Cause: one too few, or one too many brackets in an expression

eg `let x=(3+5)/7)`
should be: `let x=((3+5)/7)`

73 `Error: type mismatch`
Cause: mixing string and numeric expressions in the same expression. Expressions must consistenly contain only either string or numeric values (as represented by literals, variables and functions).

Memory and Maximum Records

The maximum number of records a file can hold is determined by the permanent memory (eg. tapes) and also the temporary memory (RAM).

Theoretical limit

ARCHIVE can store 65,535 records in a file if there is enough memory.

Tape limit

Each numeric field in a file occupies 8 bytes of space.

A byte is a measure of memory which represents (for practical purposes) the space a single character can be stored in.

Each string field occupies as many bytes as its length plus 1. You must estimate the average length of strings stored in each record.

A Microdrive has approximately 100 Kbytes (100 × 1024 bytes) of space, so the following formula gives the maximum number of records a single file on a tape can hold:

```
102,400/((nf*8)+(sf*1)+av)
```

`nf` is the number of numeric fields
`sf` is the number of string fields
`av` is the average length per record of all the string fields

Multiply the answer by about 80% (ie. 0.8) to reduce the maximum size. This is because full Microdrives can take impossibly long to save files while they look for scraps of available space.

RAM considerations

When ARCHIVE has just been loaded, there is a certain amount of space left in RAM for use by procedures, screen and data files.

In version 1, this is roughly 10 Kbytes, in version 2, 20 Kbytes.

To see for yourself type:

```
print memory()
```

This displays the amount of free memory in RAM in bytes.

Each time you load or open a file, this decreases. Screen files occupy roughly 500 bytes, and up to 500 bytes of data files are loaded at first. Program files occupy space proportional to their size. Variables also use up memory space.

The key consideration governing the maximum number of records in open files is the amount of space available for the data file indices.

Data file structure and index

When a file is opened, ARCHIVE loads not just data, but information about the file itself.

ARCHIVE sets up three areas in memory:

- file definition, including the field names
- index of gaps in the file
- index of the records

The last is the most important. ARCHIVE only knows where to find a record (which can all be different lengths, remember) because the index holds its position in the file and its length.

Index requirements

When a file is ORDERed, 8 bytes of each key field are added to the index for use by LOCATE.

Thus each level of ORDERing increases the total memory space a data file requires.

There is an upper limit in the index area of 8K (8×1024). This can be divided by the index requirements to give the maximum number of records for the file.

Each record in the file requires an index overhead as follows:

Not ORDERed	– 6 bytes
ORDERed on 1 field	– 14 bytes
ORDERed on 2 fields	– 22 bytes
ORDERed on 3 fields	– 30 bytes
ORDERed on 4 fields	– 38 bytes

The formula for the maximum number of records in a file ORDERed on 2 fields is thus 8192/14, which is 585.

If you run out of space, either reduce the number of files open at one time, or use composite keys (see Chapter 8) to reduce the number of ORDER keys.

You can reduce space required for procedures by cutting down on variables, shortening names and other literal strings, removing all REM statements, using 'arrays' and loops to avoid multiple commands, or by RUNning smaller programs one after the other (see Chapter 13).

LIMITATIONS

Maximum string length: Field – 160
Variable – 255

Mathematical accuracy – 13 digits

Largest/smallest positive number – $\pm 1.7^{\pm 38}$

Ordering – Up to 4 levels of key fields
First 8 characters of a string significant

Maximum procedures per program in memory – 255

Maximum fields in a record – 255

Field, variable and procedure names – 13 characters, the first a letter, no spaces or punctuation

String fields and variables additionally end in a '$' sign

Cannot be ARCHIVE key words

Maximum records in a file – Up to 65535 but often much less

File names – Drive names in format `mdv1_`

File names up to 8 characters, first a letter, no spaces

Extension separated by underline, up to 3 characters

Differences Between Versions

New functions in Version 2

`ATN(), COS(), DEC(), DEG(), EXP(), FIELDN(), LN(), SIN(), TAN(),`

New key words in Version 2

`OBJECT, PROTECT`

Incompatible file structure

`_scn` files

Memory available on loading

Version 1: approximately 12,000 characters (bytes)
Version 2: approximately 20,700 characters (bytes)

Quotation marks

`""` is unacceptable in Version 2

SINPUT, INSERT, AMEND, INPUT

Version 1: alpha characters in numeric fields causes error

Version 2: alpha characters in numeric field prevented

INSERT, AMEND

Version 1: **ENTER** – cycle forwards through fields
F4 – no effect
160 characters maximum line

Version 2: **ENTER** – save record if used on last field
F4 – exit/abandon
screen wide maximum line

New error message in Version 2

Error 21 – `File already exists`

ASCII Codes

The name ASCII is an acronym for a set of codes called the '**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange'.

The codes are designed to allow different makes and pieces of equipment, like computers, printers and monitors to understand each other. There are codes for numbers, letters and punctuation marks, as well as special signals like 'form feed'.

The codes are currently the international standard for micro-computers. Some mainframe and mini computers, notably IBMs, and some micros, notably Commodores, do not conform, but are the exception.

Converting ASCII codes to characters

The codes are numbers in the range 0 to 127, and can be converted into the character (including numbers 0–9) or signal they represent by using the function `CHR()`.

The function `CHR()` can also print characters for codes 128–255, but these produce different results on different computers and not part of the standard.

How they work

A computer handles all information as a stream of electrical impulses, which are either high or low, on or off. It manipulates these by using binary arithmetic, where every value can be represented by a string of 1's and 0's.

Decimal is the numbering system we use every day, representing each value with a series of numbers from the range 0 to 9.

Every binary number has a decimal equivalent:

binary		***decimal***
00000001	equals	1
10110001	equals	177
11111111	equals	255
and so on		

The letter capital **A** is stored in memory and on tape as signals representing the binary number 1000001, or decimal 65.

The command `print "A"` displays the letter `A` on the screen, so does the command `print chr(65)`. In other words, the `CHR()` function allows you to use data in a way extremely close

to how the computer itself uses it. This can be especially useful when dealing with other pieces of equipment such as printers, modems and other computers.

Converting characters to ASCII

The function `CODE()` returns the ASCII code in decimal of the first character in a string. For example:

```
print code("A")        displays the number 65
print code("Bootle")   displays the number 66
```

Thus **A** is an alphabetic or character representation of 65, which is a decimal representation of 1000001, which is a binary representation of the electronic signals actually used by the computer to represent the letter **A** in text.

Common codes

Codes which cause a letter, number or punctuation mark to appear are called *printable* codes. The main codes are:

`32 – 47`	Space and punctuation marks
`48 – 57`	Numbers `0` to `9`
`65 – 90`	Upper case (capital) letters, `A` to `Z`
`97 – 122`	Lower case letters, `a` to `z`

For example: the character **1** is stored as ASCII number 49, and the space, ' ', as number 32. Capital **M** is stored as 77, and lower case **m** as 109. The ASCII values for the same letter in upper and lower case are *always* 32 apart.

The other codes you are most likely to encounter are these:

`chr(0)`	null or nothing
`chr(27)`	**ESC** or escape

These two are used as 'control' codes in combination with others to change their meaning. They are often used for sending instructions to printers and monitors.

`chr(10)`	**LF** or line-feed
`chr(12)`	**FF** or form-feed
`chr(13)`	**CR** or carriage return
`chr(7)`	**BEL** or bell usually causes a beep noise, but is not recognised as such by ARCHIVE
`chr(26)`	**EOF** or end of file marker, also called ↑Z or **CTRL-Z**

Hexadecimal

Some ASCII tables show the numbers in both decimal and hexadecimal notation. (Hexadecimal comes from the Greek for 16: it uses 16 instead of the normal 10 (decimal) digits. 16 is more useful to a computer than 10.)

Hexadecimal notation includes the letters `A` to `F` as well as numbers, but some 'hex' numbers look just like decimal (eg. decimal `12` = hex `1B`, but decimal `65` = hex `41`). Check before using one which kind of number you are looking at.

Index

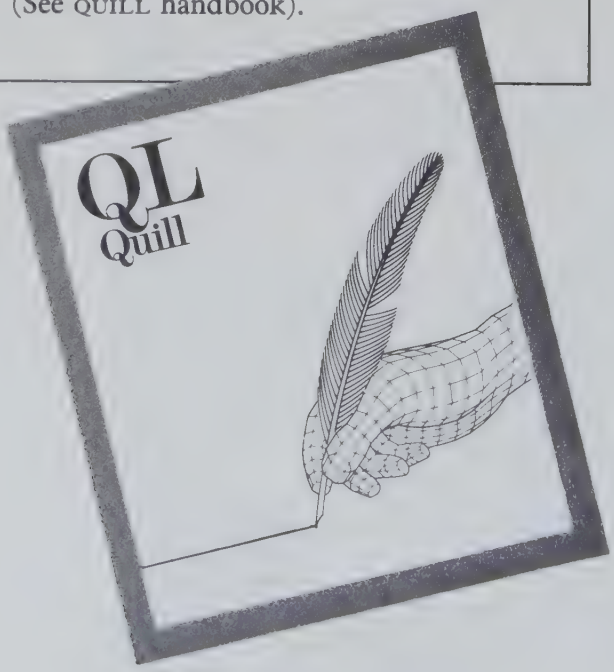
: 103
* 163
+ 163
- 163
/ 163
< 155, 156, 163
<= 156
<> 156
= 156, 163
> 155, 156, 163
>= 156
^ 163
Abacus 302
all 91, 108
alter 24, 50
and 162, 164
append 46, 109, 176
Arguments 209
Arithmetic operators 63
Array 323
ASCII code 129, 134, 181, 304
at 123
Averaging 285
back 23
backup 53, 81
Boolean logic 164
Boot 9
Buffers 136
Bugs 143
Caps lock 2
Carriage return 118
Character field 37
chr() 129
clone 5
close 51
Closing 29
cls 105
Colon (:) 103
Colour 121, 225
Commands 10, 103
Common format 302
Composite keys 182
Computer stationery 279, 312
continue 25, 149, 177
Control 2
Control area 10, 13
copy 53
Copying 338
Corrupted files 339
count() 60
CR 119
create 18, 38
Create file 18
Current file 41, 244
Current line 69
Current procedure 69
Current record 27, 44
Cut and paste 135
Data file structure 346
Date 108, 109
date() 110, 180
days() 110
dec() 294
delete 42
Delimiters 328
Differences between versions 348
dir 29
display 22, 39, 94
Display area 10
Display widths 9
Domestic TV 5, 9
Drive names 30
Dry-running 142
Dummy procedure 86
dump 310
Easel 302
edit 65
Editing text 14
else 201
endall 91
endcreate 19
endif 152, 153
endproc 66
endwhile 95
eof() 95
errnum() 196, 196
error 194, 196
Error messages 340
Escape key 2, 65
Evaluating expressions 154
export 300
Expressions 154
F1 12
F2 13
F3 13
F4 14
F5 14
false 153
Field names 37
Field variables 109

Field() 316
Fields 35
fieldt() 275
fieldv() 138
File name 31, 338
File name extensions 31
Files 35, 44, 246
find 25
first 23
Flow chart 194
Forever loops 150
Form feed 129
Format 6, 54
found() 152, 177
Free memory 346
Full colon (:) 103
Function keys 2, 12, 22
Functions 61
gen() 294
General format 210
getkey() 188
Help 12, 337
if 152, 153, 201
import 300
index 346
ink 121, 122, 123, 225
inkey() 278
input 81
insert 20, 46, 50, 176
Insert mode 71
install_bas 128
install_dat 128
instr() 159, 160, 326
Items 123
Key field 176
Keyboard 2
Keys 22
kill 107
Label 7
last 23
let 106
LF 119
Library 101
Limitations 347
Line editor 68
Line feed 119
Linking 246
llist 142
load 85, 333
Loading 338
local 287
Local variables 133, 286
locate 177, 178, 182
Logical expression 154
Logical name 40, 44
Logical operators 163, 168
Long-term storage 28
look 42, 42
Lookups 240
Loops 93
lower() 150
lprint 103, 125
lrun 5, 9
main 40
Memory 28, 345
Menus 191
merge 78, 101, 333
Microdrive cartridge 3
mode 280
Monitor 5, 9
month() 110
Nested IFs 201
new 73
next 23
not 163
Null string 134, 151
num() 294
Number fields 37, 46
Number of records 345
Numeric field 345
numfld() 141
object 333
On line 127
One-to-many relationship 236
open 29, 40
Operation priorities 260
or 164
order 174
paper 121, 121, 123, 225
Parameters 132, 133
position 93
print 60, 94, 118, 123
printer_dat 127, 128
Printers 127
proc 66
Procedure editor 68
Procedure names 189
Procedures 65, 87
Prompt 11, 207
protect 333
QL basic screen 5
Quill 300
quit 16
RAM 28
RAM considerations 345
Ranges 158
recnum() 94
Record numbers 95
Records 21, 35
Relative operators 154, 156

rem 78
rept() 204
Rescuing file 339
reset 2, 162
return 203
ROM 28
Rotating backups 55
run 78, 333
save 56, 72, 79, 86
screen 63, 94
Screen editor 224
Screen files 232
Scrolling 39
search 149, 159
Sectors 7, 29
sedit 224, 228, 229
select 160
Semi-colon 94, 118
Shift 2
sinput 253
sload 230
spooloff 307
spoolon 306, 307
spoolon "screen" 310
sprint 50, 94
ssave 229
Standard display 64, 94, 224
start 78
stop 152
str() 209
String 37, 84, 124
String concatenation 84
String slice 157, 158, 180
Structure 39
Substring 158, 325
Subtotalling 282
SuperBASIC 5
tab 120, 123
Tabulate 2
Television 5, 9
Temporary memory 28
Text editor 68
time() 110
Toggle 98
Totalling 281
trace 96, 98, 99
true 153
Unary operator 163
value() 329
Variable 82
Version 8
Volume name 7
Warnings 207
while 95, 153
Work area 10

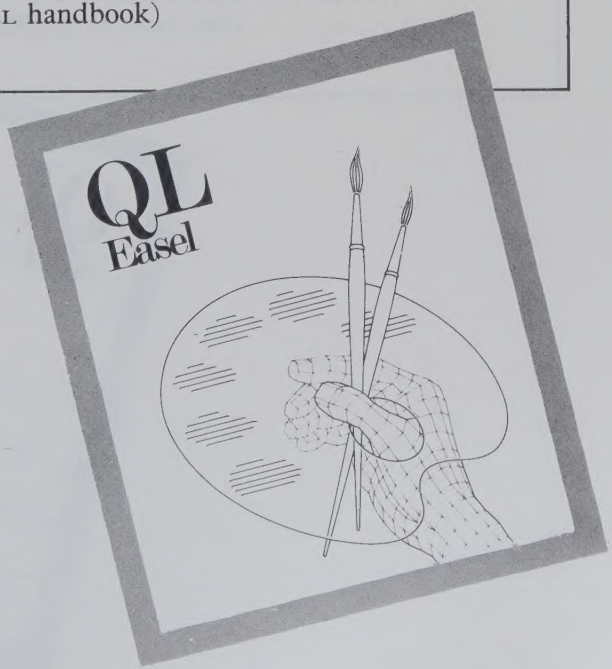


Does Joyce approve of Miranda's catch? Or does she think longingly of that nice Phillips boy? (See QUILL handbook).



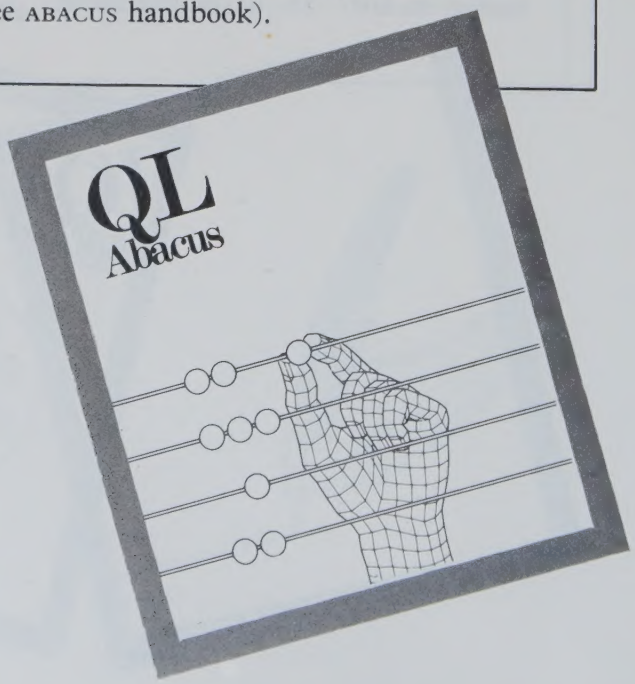


Will Nigel make the big time? Or will he end up working for his mother-in-law? (See EASEL handbook)





Will Joyce's brooches arrive in time for Christmas?
(See ABACUS handbook).



a *BLUEPRINT* handbook

QL ARCHIVE is a thoroughly practical introduction to using Archive, the powerful database program and language that is included free with every Sinclair QL computer.

The book is an excellent, straight-forward guide – each function is introduced with detailed, practical examples and sample exercises.

The procedural language that can be used to develop specialised applications is introduced and used to develop sample applications.

The friendly style of the book is suitable for those who have never used a database before and may find the prospect daunting. More experienced database users who have not used Archive before will find the book a really effective way of learning how to harness the power of Archive quickly.

In addition to teaching basic skills in using a database the book includes:

- Step-by-step instructions
- Helpful hints
- Quick reference section
- Advice on printers

QL Archive is essential reading for anyone who wants to get the most out of databases, Archive and their QL.

There are four other companion volumes in this series: *QL Abacus*; *QL Easel*; *QL Quill* and *Integrating Abacus, Archive, Easel* and *Quill.*

'An astonishing contribution to the QL – a wonderfully intelligent and creative approach to the user's needs.'
Sir Clive Sinclair

The books in this series have been produced with the cooperation and support of PSION Ltd., authors of the software.

ISBN 0-71 33-5

KO-239-383

I 5J

£14.95
ISBN 0 7126 0633 5
CENTURY COMMUNICATIONS LTD.

9 780712 606332