



PSION Xchange / PC FOUR

Quill word processor
Abacus spreadsheet

Archive database

Easel business graphics

ARCHIVE

How to use this republished manual

Text in this style is a reproduction of the original manual

Text in bold like this is what is seen or typed in the Computer

Text in this font is taken from Dilwyn Jones and others tips trick and comments

Text in this font are comments from the Author (Martin)

ACKNOWLEDGEMENTS

The text in this republished manual is taken from the QL THOR Xchange manuals distributed by Gunther Strube & Erling Jacobsen on behalf of QUANTA and downloaded from Dilwyn Jones Sinclair QL Pages. Some of the Xchange references and commands are not available in the PC FOUR version but the reader will still find the manual covers the main aspects of the package.

Contents

- Introduction to Archive 1.0
- The demonstration file 2.0
- Using ARCHIVE 3.0
- Using the commands 3.1
- The MODE command 3.1.1
- The NEW command 3.1.2
- ARCHIVE files 4.0
- File records and fields 4.1
- Files 4.1.1
- Records 4.1.2
- Fields 4.1.3
- ARCHIVE Data files 4.1.4
- Other file types 4.1.5
- Creating a file 5.0
- CREATE 5.1
- Adding records to a file 5.2
- Physical and logical file names 5.3
- Examining a file 6.0
- Displaying a record 6.1
- Examining other records 6.2
- Searching a file 6.3
- Find 6.3.1
- Continue 6.3.2
- Search 6.3.3
- Closing a file 6.4
- Modifying a file 7.0
- Insert 7.1
- Append 7.2
- Delete 7.3
- Changing a record 7.4
- Selecting records 7.5
- Sorting a file 7.6
- Sorting a Read-only-file 7.6.1
- Locate 7.7
- Closing a modified file 7.8
- Screen layouts 8.0
- Defining a screen layout 8.1
- Background text 8.1.1
- Graphics characters 8.1.2
- Screen edit commands 8.2
- Clear screen (C) 8.2.1
- Mark variable (V) 8.2.2
- Delete Variable (V) 8.2.3
- Ink (I) 8.2.4
- Paper (P) 8.2.5
- Leaving Sedit 8.3
- Activating a Screen layout 8.4
- The SPRINT command 8.5
- Saving and loading screens 8.6
- The DISPLAY command 8.7
- An example 8.8
- Procedures 9.0
- Creating a procedure 9.1
- Listing and printing procedures 9.2
- Saving and loading procedures 9.3
- Examining file records 9.4
- Editing procedures 10.0
- The line editor 10.1
- The program editor 10.2
- Selecting a procedure 10.2.1
- Selecting a line 10.2.2
- Inserting text 10.2.3
- Edit a line 10.2.4
- Editing commands 10.2.5
- Delete procedure (D) 10.2.5.1
- New procedure (N) 10.2.5.2
- Cut (C) 10.2.5.3
- Paste (P) 10.2.5.4
- Programming 11.0
- A mailing list 11.1
- Insertion 11.1.1
- Deletions 11.1.2
- Payments 11.1.3
- Changes 11.1.4
- Address labels 11.1.5
- Leaving the program 11.1.6
- Programming ERRORS 11.1.6
- The RUN command 11.1.7
- Procedure parameters 11.2
- Local procedure variables 11.3
- Prompts 11.4
- Data entry 11.5
- Text 11.5.1
- Numbers 11.5.2
- Producing a report 11.6
- Using multiple files 12.0
- Changing a record structure 12.1
- The current file 12.2
- Stock control 12.3
- The stock file 12.3.1

The suppliers file 12.3.2
The orders file 12.3.3
Enquiries 12.3.4
Stock report 12.3.5
Recording sales 12.3.6
Recording incoming stock 12.3.7
Ordering new stock 12.3.8
ARCHIVE reference 13.0
The function keys 13.1
Variables 13.2
Expressions 13.3
String slicing 13.4
Syntax 13.5
Syntax conventions 13.5.1
Syntactic entities 13.5.2
File names 13.6
ARCHIVE database files 13.7
Construction of a database file 13.7.1
Opening and closing files 13.7.2
Logical file names 13.7.3
Procedures 13.8
Print items 13.9
The program editor 13.10
Select procedure 13.10.1
Select line 13.10.2
The editing commands 13.10.3
Inserting text 13.10.4
Editing text 13.10.5
The screen editor 13.11
The screen driver 13.12
ARCHIVE commands 13.13
ALL 13.13.1
ALTER 13.13.2
APPEND 13.13.3
BACK 13.13.4
BACKUP 13.13.5
CLOSE 13.13.6
CLS 13.13.7
CONTINUE 13.13.8
CREATE 13.13.9
DELETE 13.13.10
DIR 13.13.11
DISPLAY 13.13.12
DUMP 13.13.13
EDIT 13.13.14
ENDALL 13.13.15

ENDCREATE 13.13.16
ENDWHILE 13.13.17
ERROR 13.13.18
EXPORT 13.13.19
FIND 13.13.20
FIRST 13.13.21
IF 13.13.22
IMPORT 13.13.23
INK 13.13.24
INPUT 13.13.25
INSERT 13.13.26
LAST 13.13.27
LET 13.13.28
LLIST 13.13.29
LOAD 13.13.30
LOCAL 13.13.31
LOCATE 13.13.32
LOOK 13.13.33
LPRINT 13.13.34
MERGE 13.13.35
MODE 13.13.36
NEW 13.13.37
NEXT 13.13.38
OPEN 13.13.39
ORDER 13.13.40
PAPER 13.13.41
POSITION 13.13.42
PRINT 13.13.43
QUIT 13.13.44
REM 13.13.45
RESET 13.13.46
RETURN 13.13.47
RUN 13.13.48
SAVE 13.13.49
SCREEN 13.13.50
SEARCH 13.13.51
SEEDIT 13.13.52
SELECT 13.13.53
SINPUT 13.13.54
SLOAD 13.13.55
SPOOLOFF 13.13.56
SPOOLON 13.13.57
SPRINT 13.13.58
SSAVE 13.13.59
STOP 13.13.60
TRACE 13.13.61

UPDATE 13.13.62
USE 13.13.63
WHILE 13.13.64
ARCHIVE functions 13.14
ABS() 13.14.1
ATN() 13.14.2
CHR() 13.14.3
CODE() 13.14.4
COS() 13.14.5
COUNT() 13.14.6
DATE() 13.14.7
DAYS() 13.14.8
DEC() 13.14.9
DEG() 13.14.10
EOF() 13.14.11
ERRNUM() 13.14.12
EXP() 13.14.13
FIELDN() 13.14.14
FIELDT() 13.14.15
FIELDV() 13.14.16
FOUND() 13.14.17
GEN() 13.14.18
GETKEY() 13.14.19
INKEY() 13.14.20
INSTR() 13.14.21
INT() 13.14.22

LEN() 13.14.23
LN() 13.14.24
LOWER() 13.14.25
MONTH() 13.14.26
NUM() 13.14.27
NUMFLD() 13.14.28
PI() 13.14.29
RAD() 13.14.30
REPT() 13.14.31
SGN() 13.14.32
SIN() 13.14.33
SQR() 13.14.34
STR() 13.14.35
TAN() 13.14.36
TASK() 13.14.37
TIME() 13.14.38
UPPER() 13.14.39
USR() 13.14.40
VAL() 13.14.41
VALUE() 13.14.24
Errors 13.15
Error messages 13.15.1
Modify or Create QuizPack Files 14.0
Appendix A Import, Export and Transfer
Appendix B Printer Drivers
Appendix C Zip Files contents

1.0 INTRODUCTION TO ARCHIVE

ARCHIVE is an intelligent database. You can use it to store any kind of information that you choose to type in. You are free to decide how to store and retrieve your information - you can use ARCHIVE as any type of filing system, from a card index to a full multi-file relational database. You can present information in the screen layout that ARCHIVE provides, or you can design your own layout. You can produce printed forms and reports in any format that you wish. When you have just loaded ARCHIVE it is in the keyboard interpreter mode. This means that it will accept what you type at the keyboard and try to interpret and execute it as a known command. ARCHIVE has a comprehensive set of commands which allow you to make use of its facilities from the moment that you load it. Although the commands form a powerful programming language for the construction of specialised applications, you can create a useful card index in a few minutes, directly from the keyboard. As soon as you have created a file you can use the available commands to make sophisticated searches or selections from the file, sort the records with respect to any number of fields and display the results. At all times you are guided by an informative set of prompt messages which never leave you in any doubt about what your options are or what you are expected to do. If you require further information you can always use the Help files. These contain full details about all the options. You may ask for Help at any stage, no matter what you are doing, and will automatically be given the information that is most relevant to your current needs. The real power of ARCHIVE becomes apparent when you write your own procedures in the database language. You can create a named procedure to do exactly what you want and then use it as an additional command, in exactly the same way as you use the commands provided with ARCHIVE. In this way you can write a complete program that runs independently of the normal commands. The ARCHIVE database language has a syntax similar to BASIC and is simple to learn and use. It is based on named program segments known as procedures. Using procedures leads naturally to the creation of correct and readable programs. Unlike BASIC, however, there are no line numbers. This makes it easy to build a library of useful procedures which you can include, at any position, in later programs. The mechanics of writing and modifying a program are aided by a full procedure editor which, together with the input line editor (which is available at all times), make editing a simple and painless task. The commands include simple and rapid sorting, searching and selecting records, together with many string manipulation operators and fast, accurate arithmetic. The data files themselves use variable length fields and records. Not only does this lead to the most efficient use of available memory and disk space, but also to simplified file creation. You never need to decide in advance how large a record needs to be. This manual contains a number of working examples. You can either use them immediately or you can make simple modifications to match them to your exact needs. Try out the examples to see some of the range of things that can be done. All the programs are fully documented in this manual and contain many general purpose procedures which you could include in your own programs. ARCHIVE has been

designed to give you the greatest possible flexibility. As a consequence of its open-ended programming approach it cannot give as much assistance with the selection of options as the other PC-FOUR programs. If you are not familiar with computers and computer programming it may be advisable for you to study an introductory book on programming in, for example, BASIC before attempting to write complex ARCHIVE programs.

2.0 THE DEMONSTRATION FILE ARCHIVE is supplied with a demonstration data file, "gazet.dbf", which contains a gazeteer of the countries of the world. You will find this file in the PCFOUR.ZIP. For each country the file contains the following information:



```
C:\Pision\PSION_~1\DosBox\ARCHIVE.EXE

HELP      COMMANDS  create   look     open    close
press F1  delete  display  back    alter   find
PROMPTS   first   insert   last    next    quit
press F2  type command & press ← (F3 for more)
COMMANDS
press F3
ESCAPE
press ESC

Logical name : main
country$    : AFGHANISTAN
continent$  : ASIA
capital$    : KABUL
languages$  : PUSHTU,DARI
currency$   : AFGHANI
pop         : 19.5
gdp         : 110
area       : 657

>open "gazet"
>display
>
```

In addition to being used by the tutorial programs, it is used in many of the examples for examining and modifying file records in later chapters. Before you try out these examples you should make a copy of the file onto another disk. Use this copy when you work through the examples so that there is no danger of accidentally changing the original file.

3.0 USING ARCHIVE

3.1 USING THE COMMANDS ARCHIVE's commands form a programming language, so you must type their names in full. At the main level of ARCHIVE you can press [F3] to see a further list of commands in the control area (there are four different lists). You can use the vast majority of these commands by typing the name and pressing [Enter]. (Any command which needs a file name will ask you to type it in). You can use any of the commands, even if its name does not appear in the current display in the control area. There is an example of how to use an ARCHIVE command in the following description of the mode command. Within the screen and program editors - described in later chapters - you can select an

editing command by the normal method of pressing [F3] and then the first letter of the command.

3.1.1 The MODE Command You have the option of combining the control, display and work areas into a single area by means of the mode command. The command must be followed by a number, which can be 0 or 1. A value of 0 combines the control, display and work areas into a single area. Type : mode 0 [Enter] With this form of display, your input from the keyboard and anything shown on the screen - by a command or a program - all share the whole of the screen. A value of 1 separates the screen back into its three distinct areas. Type : mode 1 [Enter]

3.1.2 The NEW Command All chapters in the ARCHIVE section of this manual assume that you start with no files in the computer's memory. If you have loaded any files you will therefore have to erase them. The simplest way to erase all the files in the computer's memory is to use the **new** command. It clears any program or files from the computer's memory, ready for a fresh start. You use the command by typing in: new [Enter] In addition to deleting all files from memory, it also clears the display area, ready for you to start again. The new command also closes any open data files before clearing the memory. Remember that you must close any open files before you remove the disk containing them from a disk drive.

4.0 ARCHIVE FILES

4.1 FILE RECORDS AND FIELDS An ARCHIVE file behaves rather like a card index. A real card index consists of a box containing a set of record cards. Each card has various items of information written on it. For such a card index to be useful, there have to be rules to determine where each piece of information is written on the card. Suppose, for example, that we have a name and address index. You would normally write the person's name across the top of the card, followed by the address and telephone number (if any). It would be very difficult to use if some cards had the name written at the top and others had it written near the bottom. You would normally expect to be able to use the index by flipping through the cards, reading only the top line, until you found the name you were looking for. If you had two sets of record cards, such as a name and address records and a set of stock records, you would not normally store them both in the same box. You would use two boxes and label them, for example, "Customer Records" and "Stock Records".

4.1.1 Files The card index system contains most of the ideas necessary to understand how an ARCHIVE file works. A file is like the card index box and is given a name to identify it. The file is made up of a collection of records, each of which serves the same purpose as a record card. A database file, then, is simply a collection of related records.

4.1.2 Records The records within a file all contain the same type of information but each record is different from its neighbours. In a customer record file, for example, each record would contain the name, address and telephone number of a particular customer, together with details of his previous dealings with your company, whether he is entitled to any discount, his credit limit, and so on.

4.1.3 Fields As in a card index, the information in each record is organised in a regular way. Some of this information might be placed on a record card, where a specified area of the card is reserved for each piece of information. A record in an ARCHIVE file is organised in the same way. Each item is stored in a separate region of the record, known as a field. A record in a customer file, such as that described above, would contain a name field, an address field, a discount field and so on.

4.1.4 ARCHIVE Data Files If this were the whole story there would be little point in using an ARCHIVE data file in preference to a physical card index. There are, however, many advantages in using computerised records. A customer record card index would normally be arranged in alphabetical order of customer names which makes it an efficient way to find the information about a particular customer. Suppose, however, you want to send a letter to all your customers who have not placed an order with you during the last six months. It would be a very tedious task to go through the entire contents of a card index to compile such a list. In ARCHIVE you can make such a search by using a few simple commands. Furthermore, it is easy to arrange for a set of address labels to be printed at the same time.. *Example 11.1 Mailing List shows you can save a great deal of time and effort by using ARCHIVE to store and manipulate your records.*

4.1.5 OTHER FILE TYPES ARCHIVE uses other types of file as well as data files. Each different type is assumed to have a particular file name extension. Unless you specify a file name extension yourself, ARCHIVE will assume the standard extension for that type of file. All data files, for example, are saved and loaded with an assumed extension of **.dbf**. There are four main groups of additional files. These are: Program files - used to store programs written in the ARCHIVE language. Unless you specify otherwise ARCHIVE assumes that they have a file name extension of **.prg**. There is also an option to use program files stored on disk in ARCHIVE's internal format, rather than the normal plain text format. They are faster to load and save since they do not have to be translated from one form to the other in the process. ARCHIVE assumes that such files have an extension of **.pro**. Screen layout files - used to contain your own design of screen layout for the display of your data file records. ARCHIVE assumes that such files have an extension of **.scn**. Import and export files - used to transfer information between ARCHIVE and the other programs in the Psion PC-FOUR family. It is assumed that such files have an extension of **.exp**. Print files - used to store printed output for later printing with an assumed extension of **.lis**. Print files created by the dump command have an extension of **.dmp**.

5.0 CREATING A FILE Suppose you want to use ARCHIVE to make a catalogue of your books. To do this, you will have to create a new file called, for example, "books". The first thing to do when creating a file is to decide what information it is going to contain, that is, what fields you will use in each record. In this case you will obviously need to record the author, title and subject; you may also like to include other details, such as the type (fiction or non fiction), ISBN (International Standard Book Number) shelf location, a brief description and so on. In this example we shall simply use three text fields to contain the author, title and subject and one numeric field which will be used to hold the ISBN.

5.1 CREATE You create a file by using the create command. You must specify the name of the file to be created and the names of the fields to be used in each record (the names of fields which are to hold text strings must end with a dollar sign). When you have finished defining the fields of a record you end the create command with endcreate. You can create a simple book catalogue file, as described above, by typing in the following sequence. (From now on we shall not always show the [Enter] that you must use at the end of every line of input.)

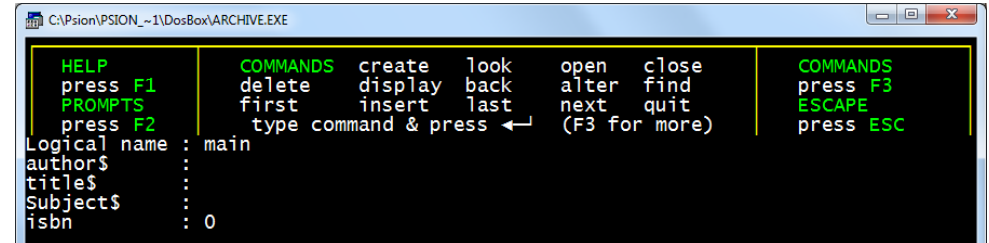
```
create "books" [Enter]
author$ [Enter]
title$ [Enter]
subject$ [Enter]
isbn [Enter] [Enter]
```



```
>new
>create "books"
author$
title$
```

Note that you do not have to type in the final endcreate command. You can do so if you want, but you can end the creation of the file simply by pressing [Enter] on a blank input line. (You must, however, include endcreate if you use create in an ARCHIVE program.) One of the great advantages of ARCHIVE is that you do not have to decide in advance how much memory is to be reserved for each field within a record. If you had to decide the length of each field at the time the file was created you would have to allow for the longest possible record that you would expect to appear. This would mean that a record containing less than this maximum amount of information would have a lot of wasted space, reducing the number of records that you could keep. ARCHIVE allows each field within a record to be of variable length; the space used for each field is automatically adjusted to match the amount of information stored in it. There is no need to decide in advance how much space should be reserved for each field. This makes it much easier to create a file and also ensures that the computer's memory is used to maximum efficiency. ARCHIVE can accept fields as large as 255 characters. Although it is quite straightforward to change the fields used in a file, it is worth taking a little care in deciding what fields to use before you create the file.

5.2 ADDING RECORDS TO A FILE When you have created a file as described above it is in the computer's memory as a file which is open for both reading and writing, but as yet it contains no records. There are several ways to add records to the file and they are fully described later. The simplest way is to use the insert command as in the following example, which uses the book catalogue file that we have just created. Type: **insert** [Enter] The display area will now appear as shown below, with the names of the fields listed and the cursor positioned at the first field.

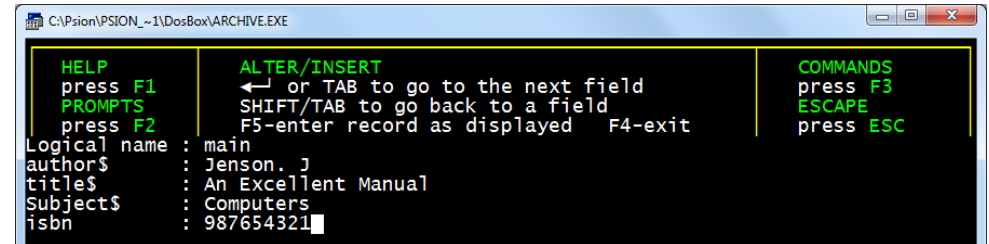


```

HELP      press F1
PROMPTS  press F2
Logical name : main
author$   :
title$   :
Subject$  :
isbn     : 0
COMMANDS create  look      open  close
          delete  display  back  alter  find
          first  insert   last  next  quit
          type  command & press ← (F3 for more)
COMMANDS press F3
          ESCAPE
          press ESC

```

All you have to do is to type in the contents of each field. You step from field to field by pressing either [Enter] or [Tab]. You can also use [Shift] and [Tab] together, to move back to the previous field. For example, type in the following entries, but please do not press [Enter] after typing in the last value: Jensen, J [Enter] An Excellent Manual [Enter] Computers [Enter] 987654321 The display area should now appear thus: logical name : main author\$: Jensen, J title\$: An Excellent Manual subject\$: Computers isbn : 987654321 Ignore, for the moment, the first line of the display. It will be explained in the following section.



```

HELP      press F1
PROMPTS  press F2
Logical name : main
author$   : Jensen, J
title$   : An Excellent Manual
Subject$  : Computers
isbn     : 987654321
ALTER/INSERT
          ← or TAB to go to the next field
          SHIFT/TAB to go back to a field
          F5-enter record as displayed F4-exit
COMMANDS press F3
          ESCAPE
          press ESC

```

Each line represents a field of the record. It is made up of two parts; the name of the field and the a value (eg. the text "Jensen, J") which will change from record to record and a name (eg "author\$"). If the field is to contain text its name must end with a dollar sign otherwise it will be a numeric field. ARCHIVE will accept any valid number or constant numeric expression (such as 3+9) as the contents of a numeric field. The name of a field is sufficient to identify its value if there is only one data file open. IMPORTANT : In this display format each field occupies a line on the screen so it is not possible to show more fields than there are lines in the display area. If you create a file with many fields per record you will have to design your own screen layout in order to show them all on the screen. This is explained in detail in the chapter on Screen Layouts.

The record that you have typed in is inserted in the file when you press [Enter] after entering the value of the last field. You may also insert the record at any time, without necessarily entering values for all the fields, by pressing [F5]. In either case ARCHIVE clears all values you typed in, ready to receive the next record. If you do not wish to add more records to the file, press [F4] to leave the insert command. Note that anything you have typed since the last time you added a record to the file will be discarded. Press [F4] now to leave insert. The file contains just one record which remains displayed on the screen. Each field name, in addition to identifying a field, also behaves like a variable, that is, a memory location whose contents may be changed. The contents of these memory locations are automatically modified by the values of the fields of the current record. As you move from one record to another the variables take on the values of the fields of the new current record. You may also change the values in these memory locations yourself with the `let` command. Try typing: `let isbn = 123456789` [Enter] The contents of the variable and the display both change to the new value. The field in the file record, however, does not change and it still contains the original value of 987654321. It is not altered unless you use the `update` command. This forces each field of the current record to take the value of the variable with the same name as the field. It is dealt with in more detail later in this manual. You must remember, before you switch off the computer, to make sure that the file contents are saved on the disk by using the **new** command.

5.3 PHYSICAL AND LOGICAL FILE NAMES Every file is saved on disk with a unique name, known as its physical file name. An ARCHIVE data file must also be given a logical file name at the time it is opened. ARCHIVE uses logical file names to distinguish between data files when two or more are open at the same time. You are free to choose this name yourself at the time you open or create a file. Suppose you want to create a file called "book2" and use a logical file name "second". You can do this by writing the first line of the create command as: `create "book2" logical "second"` The rest of the command follows exactly as before. The logical file name is always shown in the first line of the simple form of the display of a record. If you do not specify the logical file name, as in the previous example, ARCHIVE will automatically supply the logical file name, "main", when you open or create a file. The logical file name of each open file must be unique. You cannot therefore open or create another file without including a logical file name after ARCHIVE has given the name "main" to a file. If you attempt to do so ARCHIVE will show the message: **MULTIPLE FILES - ENTER LOGICAL NAME** and will wait for you to enter a name. ARCHIVE refers to a specific file by its logical file name and not its physical file name. This means that you can write a single program that will work with several different files, provided that the files all have the same structure. For example, you could use a program that adds, deletes or modifies records in, say, two name and address files - one for your friends and one for your business contacts. The examples in the chapter on Multiple Files illustrate how you use logical file names to distinguish one of several open files.

6.0 EXAMINING A FILE Before you can use a data file you have to open it - this makes its contents available to you. There are two commands which you can use to open a file; `look` and `open`. The `look` command opens a file in such a way that you may only read its contents - you are not allowed to add or delete records or to alter them. This therefore removes any risk of accidentally destroying any of your data. It also enables several tasks to open the same file simultaneously. All tasks must then use `look`. The same file can also be opened more than once in the same task, by using different logical file names. The `open` command allows you to read and alter the contents of the file. It is described in the next chapter. Suppose you want to examine the "gazet" data file, supplied with ARCHIVE and described in an earlier chapter. Make sure you have a copy of this file on the default data drive. You can then open the file for reading by typing:

```
look "gazet" [Enter]
```

The `look` command does not display any file records. If, for example, you want to examine the "gazet" file in the way just described, but would like to use the logical file name "g", you can do this by using the `look` command in the following way: `look "gazet" logical "g"` [Enter] This opens the file for reading only, as described earlier, but with a logical file name "g" instead of the default name "main". All the commands described in this chapter can be made to act on a specific file by adding an optional logical file name. This will mainly be of use when you are using more than one file. When you only have a single file it is not necessary to give the logical file name in a command, even if you have specified one at the time you opened the file. All these commands will act on the current file, regardless of its logical file name if the optional logical file name is not included. If you only have one file open it is, of course, always the current file. The idea of a current file and the use of more than one file are described in the chapter 12.0 on Multiple Files.

6.1 DISPLAYING A RECORD To show the first record you type:

```
first [Enter]
display [Enter]
```

This produces a display of the first record of the file as a list of the field names and their contents, using the same format that we saw for the `insert` command in the previous chapter. The `display` command always uses this form, regardless of any special screen layout that you have designed (see the chapter Screen Layouts). If you load a designed screen layout and then use the `display` command, the layout is replaced by the simple list. If, after using `display`, you want to use your own layout again you will have to load it once more from the disk. The reason for this behaviour is that it allows you to display your file records in a simple way, without first having to design a display screen. You can always show a file record in this form by typing the command:

```
display [Enter]
```

6.2 EXAMINING OTHER RECORDS Having looked at the first record of the file, you may want to move to the following record. You do this very simply with the `next` command - type: `next [Enter]` It moves to the next record in the file. When you are typing single commands after a display command the display area is continuously updated to show the contents of the current record. You can use the `next` command to step through the file, record by record, until you reach the end of the file (it will not pass the last record). A quick way of repeating the last command that you typed in is to press [F5] and then [Enter]. When you press [F5] ARCHIVE puts the text you last typed back into the input line, exactly as if you had typed it in again. If you are using your own choice of logical file name you can, for example, move to the next record in the file by giving its logical file name after the `next` command:

`next "g" [Enter]`

There are three other related commands which you can use to control which record of the file is displayed. These are:

back - which moves to the previous record,
first - which moves to the first record,
last - which goes to the last record of the file.

Try using these commands to move around the file, displaying any record you like. Note that the four commands **first**, **last**, **next** and **back** do not themselves display the record. They merely move from record to record. You will only see the contents of the record if you have previously used the display command (or have produced an active screen layout, as described in the chapter Screen Layouts).

6.3 SEARCHING A FILE The commands described so far allow you to examine any record within a file and search through the file record by record to find the information you want. This technique is quite suitable for searching through a file with only a small number of records but would be very inefficient on a large file. The commands described in this section will allow you to make such searches automatically, selecting any one or more records that you require.

6.3.1 Find The first and simplest command is `find`. This will search from the beginning of a file, looking for the first occurrence of a specified piece of text in any of the text fields. For example:

`find "africa" [Enter]`

When you press [Enter] there will be a slight pause and then the first record containing the word "africa" in any of its text fields will be displayed. Note that this search is independent of whether the text is in upper or lower case and will therefore find "Africa", "AFRICA", "aFrIcA" or "africa". If ARCHIVE fails to find a record which matches the text it will not move from the record which was the

current record when you used `find`. You can always find out if the search has been successful by examining the value returned by the function `found()`. The value is either one or zero, depending on whether a record was or was not found. Try this by typing:

`print found() [Enter]`

after using `find` in a situation where it succeeds and one where it fails (eg `find "xxxxx"`). The `print` command displays on the screen the value of any following expression. Note that, when used directly from the keyboard, it will delete any display of a file record before printing the value. You will need to restore the display of the record (for example, with `display`) before continuing. This will not, however, be necessary when you use a `print` command in a program.

6.3.2 Continue If the first record that is found containing the text is not the one that you want, you can find the next occurrence by typing:

`continue [Enter]`

The `continue` command will repeat the previous `find`, looking for the next occurrence of the text in any text field of the following records. If, at any stage, no match is found in the remaining records of the file the display will keep the last record shown and the value returned by `found()` will be zero. It is possible that you may have to repeat a search several times before you find the record you want. Remember that, if you want to use a command repeatedly, you can use [F5] to bring back the last text you typed in.

6.3.3 Search A second method of locating a particular record is to use the `search` command. Search must be followed by a condition which results in a numeric value. The records of the file are scanned for the first one in which this number is non-zero. This allows you to find a record by specifying the contents of one or more specific fields, for example:

`search continent$="EUROPE" and languages$="FRENCH" [Enter]`

will find the first record in the file which matches both conditions. Unlike the `find` command, which examines all text fields of the records, `search` will only test the fields you specify. Also, it is case-dependent - ie it distinguishes between "EUROPE", "Europe" and "europe". If you want a case-independent search you can use either of the case-changing functions `upper()` or `lower()`. For example:

`search upper(continent$)="EUROPE" [Enter]`

This example converts the contents of the `continent$` field to upper case before making a comparison. The result is therefore independent of whether the `continent$` field contains upper or lower case text, or any combination. Again you

can use the `continue` command to find the next occurrence, if the first is not the one which you want. As with `find`, the value returned by `found()` will tell you if the search was successful. It may not be obvious that the conditions following search in these examples result in numeric values. A logical condition such as `continent$="EUROPE"` is different from an assignment, eg `let continent$="EUROPE"`. In the assignment we are just giving the variable `continent$` the value "EUROPE". All assignment statements must start with `let`. The condition, however, tests if the variable `continent$` already has a value equal to "EUROPE". If it has, the condition results in a value of one, showing that the condition is true. If it is false, the result is the value zero. Type in the following example which uses both assignments and logical conditions. It should help to make the difference clear.

```
let x = 10 [Enter] (assignment)
print x [Enter] (prints 10, the value of x)
print x =10 [Enter] (prints 1 because x has the value 10)
print x =11 [Enter] (prints 0 because x is not equal to 11)
```

A logical condition always results in a value of 1 (true) or 0 (false) depending on whether the condition is satisfied or not. Several logical values can be combined with either **and** or **or**. If two values are joined by **and**, the final result is true only if both of the logical values are true. Combining two logical values with **or** results in a true value if either of the two is true. Although the result of such a logical condition can only result in a value of 1 or 0, ARCHIVE will accept any non-zero value as meaning true.

6.4 CLOSING A FILE So far, when we have finished using a file, we have closed it with the `new` command. This command is rather drastic, as it also erases all the files in the computer's memory. An alternative way is to use the `close` command. This only acts on a data file, leaving any program, or screen layout, intact. Finally, if you have finished using ARCHIVE, you can quit. This command closes all open files and deletes the current ARCHIVE task before closing. Remember that you should never remove a disk from a disk drive while it contains open files.

7.0 MODIFYING A FILE The `open` command allows you either to read the file or to write new information to it. You should not, in general, use the commands described in this chapter with a file opened with `look`. If you attempt to do so you are given an error message to indicate that you are attempting to modify the file. There are some exceptions which are described later. If you open a file with the `open` command, instead of the `look` command you will be able to write to the file (ie change its contents) as well as reading it. This means that any additions, deletions or modifications will make a permanent change to the copy on the disk. You should therefore work with a copy of the "gazet" file, rather than the original, when you try out the examples in this chapter. As with `look`, you have

the option of specifying your own logical file name when you `open` a file, as shown in the following examples.

```
open "gazet" [Enter]
open "gazet" logical "g" [Enter]
```

Display the first record of the file with:

```
first [Enter]
display [Enter]
```

7.1 INSERT The `insert` command is used within the context of modifying an existing file, in exactly the same way as we have seen when creating a file. Remember, you can use `[F5]` to enter a record to the file at any time, even if you have not typed in values for all the fields. New records inserted into a file are normally placed at the end of the file to maintain historical order. If however, the file has been sorted, ARCHIVE inserts the new record at the correct position to maintain the sorted order.

7.2 APPEND A second method of adding a record to the file is with the `append` command. This makes a new record whose fields are filled by the current values of all the field variables for that file. Before using `append` you should therefore give the field variables the values you want them to have, eg by using the `let` command:

```
let continent$ = "AFRICA" [Enter]
let country$ = "FRANCE" [Enter] etc.
```

Any field variable that you do not give a value retains its current value. If you then type:

```
append [Enter]
```

ARCHIVE adds the new record to the file. As with `insert`, the position in the file where the new record is inserted will depend on whether the file has previously been sorted or not.

7.3 DELETE If you want to remove a record from the file, you can do so by using the `delete` command. `Delete` removes the current record (the one shown by the `display` command) from the file. All you have to do to remove a particular record is to display it, and, having made certain that it is the correct one, type:

```
delete [Enter]
```

7.4 CHANGING A RECORD It is also simple to modify the contents of any or all of the fields within an existing record. There are two methods. The first is to use the **alter** command. Select the record you want to change (eg by displaying it)

before you use alter. It works in the same way as insert, except that each field shows its old contents. You can step over (by pressing [Tab] or [Shift] and [Tab]) those fields whose contents you do not want to change. Type in a new value, or use the line editor to modify the old one. As with insert, you are allowed to put only valid numbers into a numeric field. When you have made all the changes you want, press [F5] to replace the old record with the new one. As with insert, the new version of the record is also added to the file when you press [Enter] after typing a value for the last field in the record. If you decide you do not want to change the record, press [F4]. This leaves alter immediately, without the changes you have typed since the last addition of a record having any effect on the file. The second method is by using update. As with alter, you first select the record which you want to change (eg by displaying it). You then change the contents of the field variables - for example, with the let command - until the displayed record is exactly as you want. Finally type the command:

update [Enter]

Suppose, for example, that you decide that Iceland should be classified as being in Europe, rather than the Arctic. You can modify this record as follows. First locate it by typing:

find "iceland" [Enter]
display [Enter]

Then use the let command to change the contents of the continent\$ field:

let continent\$ = "EUROPE" [Enter]

Finally, put this change into the record by typing:

update [Enter]

In both of the above methods the new record will be inserted in the correct position if the file has been sorted. Otherwise the replacement record is inserted at the end of the file. The alter command is simpler to use, but always affects the current record. The append command allows you the option of specifying the logical file name of the file you want to affect, regardless of the current file.

7.5 SELECTING RECORDS In many cases, you may want to look at a sub-group of the records within a file. Suppose, for example, you only want to look at the details of countries in Europe. You can use the select command to pick out from the file all those records which satisfy a certain condition. The file will then behave as though only those selected records are present. Try this command on the "gazet" data file to see how it works. First type:

print count() [Enter]

which will tell you how many records there are in the file. Then print the value again, after using a select command, for example:

select continent\$="EUROPE" [Enter]
print count() [Enter]

and you will see how many records are left in the file. You can, by using select again with another condition, remove further reduce the number of records in the file. There is no limit to the number of times that you can repeat select commands. The records that are removed from the file are still held in the computer's memory. You can restore them to the file at any time by using the reset command. Type:

reset [Enter] and **print the value of count()**

again, to check that the file has been restored to its original state. After a reset command, the next select will start with all the records in the file. You can combine more than one condition in a single select command, with the aid of and or. For example:

select continent\$="AFRICA" and instr(languages\$,"FRENCH")

which selects all French-speaking African countries. Note that we have used the instr() function to ensure that we include countries in which languages other than French are also spoken. The instr() function searches for the second piece of text in the first. In the above example it searches for the text "FRENCH" in the text of the languages\$ field. If the text is found it returns the position - as the number of characters from the start - or zero if it is not found. If we had used the command:

select continent\$="AFRICA" and languages\$="FRENCH"

we would have selected countries which spoke only French.

7.6 SORTING A FILE In an unsorted data file the records are in historical order, ie next steps through the file records in the same order as you originally added them to the file. This may not always be the order you want, so you will need to sort the records into the required order. You can use the order command to sort the file by the contents of numeric or text fields. You may order a file opened with look, but there are some restrictions in this case - see later. Suppose, for example, you want to sort the records of the "gazet" file alphabetically by capital. You can do this by using the order command as follows:

order capital\$a [Enter]

The 'a' following the semicolon specifies that you want to sort the file in ascending order. Replace it by 'd' if you want the file sorted in descending order. The capital\$ field becomes the sort key for the file. Only the first eight characters

of text are taken into account by order. You can specify up to four fields on which to sort the file - by giving a list of fields after the order command. For each of the fields you must specify whether the sort is to be in ascending or descending order. The following command, for example, will sort the "gazet" file into ascending (alphabetic) order by continent and descending order by population.

```
order continent$a,pop;d [Enter]
```

Note that a semicolon separates each field name from the "a" or "d" that specifies ascending or descending order, but that each pair (field name and letter) is separated from the next by a comma. The combination of fields becomes the sort key for the file. When more than one field is specified for sorting purposes the records are initially sorted according to the contents of the first field in the list. If two or more records have the same contents for this field they are ordered according to the next field in the list. If records exist which are equal in respect of the contents of both of these two fields they are ordered according to the contents of the third field, and so on. The order command always starts with all the records in the file. It can, therefore, take a long time to order a large data file. On many occasions you may only want a subset of records to be ordered and it would be wasteful to order the whole file and then select the few records you want.

7.6.1 Sorting a Read-only File You are never allowed to make permanent changes to a file opened with look. ARCHIVE will not allow you, for example, to add or delete records in such a file. You may, however, use order on a file opened with look. If the same file is opened with look more than once using different logical file names, each logical file can be ordered in a different way.

7.7 LOCATE When a file has been sorted, you can use the locate command to locate a particular record. Locate is followed by an expression (see section 13.3 for the meaning of 'expression') and finds the record whose primary sort key is equal to the given expression. If there is no record that is an exact match then locate will find the following record in the ordering sequence. For example, if the "gazet" file has been sorted by:

```
order pop;d,capital$a then the command:  
locate 100 [Enter]
```

locates the first record in the sorted file which has a population of 100 (million). If there is no such record ARCHIVE will locate the first record with a population less than that figure (remember that the file was sorted in descending order of population). The expression may be either text or numeric, but must be of the same type as the field used to sort the file. You can locate a record with respect to the contents of more than one sort key by using locate with multiple expressions, separated by commas. For example, if the file is ordered by the command shown above, then:

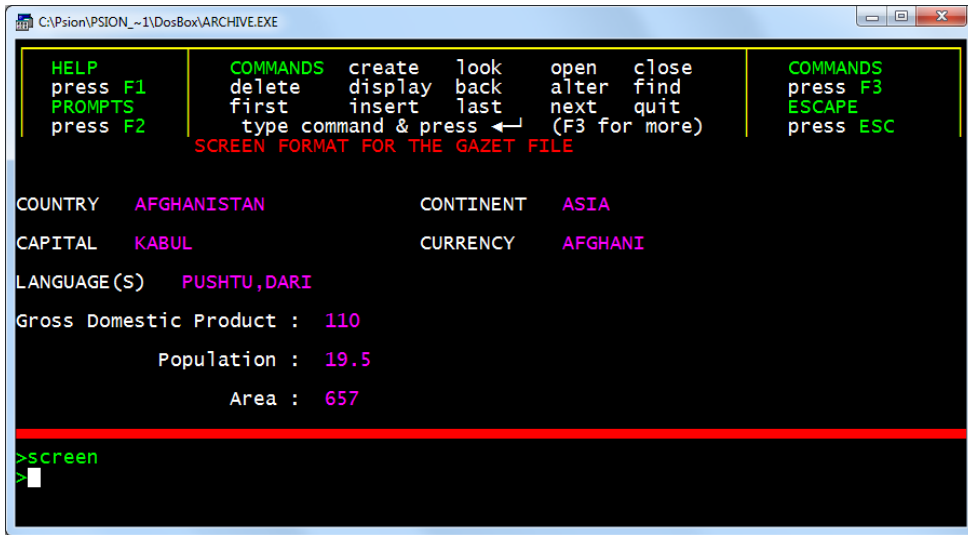
```
let a = 100 [Enter]  
let b$ = "D" [Enter]  
locate a,b$ [Enter]
```

will find the first country with a population figure of 100 or less and with a capital whose name either starts with "D", or is after D in the alphabet. In this example, ARCHIVE will locate Bangladesh, with a population of 76.1 million and capital Dakar. The only restriction on the number of expressions that you can use with locate is the number of fields used to sort the file. As with find and search, locate affects the value returned by the function found(). Provided a record is found that exactly matches the specified condition, then found() will return a value of 1 (true). Note that, as explained above, it does not search through the records one by one, but uses the knowledge that the records are sorted to go straight to the record that matches or exceeds the given condition. It will therefore always locate a record in the file, whether or not there is an exact match. If the located record is not an exact match to the specified condition then found() returns a value of zero (false). You cannot use continue after locate, since repeating a locate with the same condition will always locate the same record. You would use locate as the fastest way of locating a record in a large, sorted, file. Because of the uncertainty in the record that is located, you will usually have to make a further check on the record to make sure it is what you want - eg:

```
if found() print "Match" else print "Match failed" endif
```

7.8 CLOSING A MODIFIED FILE After making any modifications to a file you must close it. This makes sure that the changes are recorded on the disk. If you do not close a file properly (for example, if you just turn off the computer when you have finished) ARCHIVE will not be able to make sure that all your changes are recorded on the disk. Your most recent changes will therefore not be present when you next use the file. Furthermore, it is possible that your file could be left in a corrupted - and therefore unusable - state. Always make sure that there are no open files on a disk before you remove it from the disk drive.

8.0 SCREEN LAYOUTS When you use the display command on a file that you have created yourself, as in the earlier book catalogue example, the records are shown in a simple form. The logical file name is shown at the top of the screen, followed by a list of all the field names in a record of the current file. The current value of each variable is displayed to the right of its field name. You can also design your own screen layout, better suited to the information in your data file. It is very simple to produce a screen layout of your own.



8.1 DEFINING A SCREEN LAYOUT You select screen editing with the `sedit` command - type in:

`sedit` [Enter]

The display area shows the current screen layout, if any. If you have used `display` (since the last time you used `new`) the layout will be the one that ARCHIVE creates automatically. Alternatively the screen may show another layout or, if there is no screen layout in the computer's memory, the display area may be blank. If a layout is shown, you will see that the values of the fields of any file are not included. The spaces where these values are normally shown are marked by rows of dots. You should think of a screen layout as a background, against which the values of a number of variables are shown in specific positions. ARCHIVE shows a screen layout in two stages - first it draws the background text and then it shows the values of the variables at the marked positions on the screen. When you have just selected `sedit` you are at the main level of the command and you have three options: type background text into the screen press [F3] to use a screen editing command press [Esc] to leave `sedit`

8.1.1 Background Text You can move the cursor to any point in the display area by using the four cursor keys. Anything that you type will immediately appear in the display area at the position of the cursor, becoming part of the background of the layout. Try pressing the [Shift] key and, while holding it down, pressing one of the four cursor keys. The last character you typed is repeated and the cursor moves one character in the direction corresponding to the cursor key. If you keep the keys pressed the character will be repeated, forming a line. This is useful for drawing lines or borders in your layout, particularly if you use the graphics characters described in the next section. The only exception when typing

background text is if the cursor is positioned within an area of the screen reserved for the display of a variable. In such a case ARCHIVE shows the name of the variable in the work area at the bottom of the screen. You cannot type background text into this area unless you first free the area, as described later.

8.1.2 Graphics characters Although I know this works in QL-Archive it doesn't seem to work in PC-FOUR Archive. A set of 11 box-drawing characters is available in the screen editor. To draw a graphics character in a screen layout, at the current cursor position, press [F5] followed by one of the keys from 'a' to 'k' inclusive. The resulting characters are shown below

[F5] followed by: a b c d e f g h i j k l
gives | +- +- +- +- +- +- --- +- +- +- |

8.2 Screen Edit Commands There are four screen editing commands. You select one of these commands by pressing [F3] and then the key shown in brackets.

8.2.1 Clear Screen (C) When you create a screen layout, anything that appears in the display area will be part of the screen. It is probably therefore a good idea to start by clearing the display area entirely. Press [F3] and then the C key. ARCHIVE asks you to press [Enter] to confirm that you really want to clear the screen. If you press any other will cancel the command and ARCHIVE will return you to the main level of `sedit`.

8.2.2 Mark Variable (V) Suppose you want to show the value of the variable `name$` at a particular position in the screen. Move the cursor to that point and press [F3] and then the V key. ARCHIVE asks you to type in the name of the variable. You type, for example:

`name$` [Enter]

Note that this name does not appear in the screen - you are just marking the point where the value is to be shown. When you press [Enter] ARCHIVE asks you to mark how much space is to be reserved for showing the value. Press the space bar to mark the space with a row of dots. Pressing the down cursor key reserves space on following lines, converting the reserved space to a rectangular area. You can also use the left and up cursor keys to reduce the size of the box. The ability to reserve space on more than one line allows strings of up to 255 characters to be input directly via the commands `sinput`, `alter` and `insert`. When you have reserved enough space press [Enter] and ARCHIVE takes you back to the main level of `sedit`. If you move the cursor into one of the reserved areas (marked by dots) ARCHIVE shows, in the work area, the name of the variable for which space is reserved.

8.2.3 Delete Variable (V) If you press [F3] and then V - to reserve space for a variable - in a region which overlaps any area that is already reserved, you are given the option of cancelling the old area. Once you have cancelled the space reserved for a variable, its value will not be shown in the screen layout. You can, if you wish, use the option again to allocate the space to a new variable.

8.2.4 Ink (I) Suppose you want to change the ink colour to red. Move the cursor to the point where you want the red text to start and press [F3] and the I key. ARCHIVE shows the four available colours in the control area (ARCHIVE initially selects white ink). To change the colour you press any key except [Enter] until the colour you want - red in this case - is shown. You then press [Enter] to record your selection and ARCHIVE takes you back to the main level of *sedit*. Anything that you type now appears in red. The colour you select remains in effect until you use the ink command to change the colour again.

8.2.5 Paper (P) Select the command by pressing [F3] and then the P key. As with the ink command, you press a key until the paper colour you want is shown in the control area - then select that colour by pressing [Enter]. ARCHIVE initially selects black paper. If you want a change of colour to affect only part of a line, you should move the cursor to the start of the region and select the paper and ink that you want. You should then move the cursor to the end of the region and make a second selection of paper and ink, returning them to their original values.

8.3 Leaving Sedit When you have completed the display screen design to your satisfaction you should press [Esc] to leave the *sedit* command. Use *ssave* to save your screen layout on disk.

8.4 Activating a Screen Layout Once you have designed a screen layout and have left *sedit*, the screen layout will be left in the active state. This means that the values of all the variables in the screen layout will be displayed automatically every time that ARCHIVE completes a command (or a program). If, for example, you type the command *next* ARCHIVE moves to the next record of the current file and shows those fields that are included in the screen layout. If a screen layout is in the computer's memory but is not active, you can activate it with the *screen* command. This displays the background text of the screen layout, but does not show the current values of the variables. When you are in the keyboard interpreter mode (ie you are not running a program) a screen which you have previously designed and saved on a disk is also left in an active state when you load it into the computer's memory with the *sload* command, described in a later section. In a running program you must use *screen* to activate the layout after an *sload*. This allows the program to pre-load a screen layout without immediately affecting the screen contents. Any active screen is deactivated each time you use the *cls* command.

8.5 THE SPRINT COMMAND ARCHIVE will not automatically update an active screen layout from within a program. Suppose you want to show all the

records of the current file, one after another, and tried to do so by typing the one-line program: **first: let x =0: while x<count(): next: let x =x+1: endwhile** [Enter] (The *while* and *endwhile* commands cause the section of program that they enclose to be performed repeatedly, while the condition following *while* is true, ie non-zero. For correct operation every *while* command must have a matching *endwhile*.) This program would fail to do what you want, since ARCHIVE only updates the contents of the screen layout at the end of the program. You can, however, force a display of the values of the variables in an active screen from within a program with the *sprint* command. The following one-line program will show all the records, as required. **first: let x=0: while x<count(): sprint: next: let x=x+1: endwhile** [Enter] If there is no active screen you will find that *sprint* has no effect. Note that this is a rather artificial example, designed to make a special point. It would be simpler and more efficient to produce the same effect by using: **first: while not eof() : sprint: next: endwhile** [Enter] or even: **all: sprint: endall** [Enter]

8.6 SAVING AND LOADING SCREENS You can save your screen design on a disk by using the *ssave* command: **ssave "filename"** [Enter]

where "filename" is the name of your choice. The screen layout is saved exactly as it appears. You can reload the screen layout at any future time by typing in the command:

sload "filename"

When you load a screen layout directly from the keyboard it is automatically displayed on the screen and made active. From within a program the screen layout is not automatically displayed and activated by *sload*. You can load the screen layout at the start of a program and then use the *screen* command to display and activate it at a later time.

8.7 THE DISPLAY COMMAND Once you have an active screen layout you can use all the display words (*first*, *last* and so on). The current values of any variables in the screen layout are displayed automatically at the end of a command or a program. Remember that the *display* command uses its own layout. It will always replace any screen layout with its own simple list of the fields of the current record of the current file. You must therefore **ssave** your screen layout before you next use *display*. If you do not, your screen layout will be replaced by that used by *display* and you will not be able to get it back again (unless you redesign it with *sedit*).

8.8 AN EXAMPLE It is important that you are sure about the distinction between the text that is shown in the screen background and the value of a variable. The following example may help to make this clear. Suppose you want to label an area of the screen with the word "Name", and reserve a 15 character

space following the label to display the value of the variable name\$. You should move the cursor to the place where you want the label to start and type it into the screen exactly as you want it to appear. Next you should press [F3] and the V key, and then type in the name (ie name\$) of the variable you want to display, ending it by pressing [Enter]. Finally you should press the space bar 15 times and press [Enter] again. That region of the screen will appear as:

Name: Leave the screen editor by pressing [Esc].

Now give the variable, name\$, a value - for example:

let name\$="Hans" [Enter]

As soon as you press [Enter] the value is shown in the screen layout. Try using the let command to change the value of name\$. You will normally use the screen layout to show the fields of the records in a data file. However, as this example demonstrates, you can use the layout to show the values of any variables, including those you use in a program.

9.0 PROCEDURES The commands and functions of ARCHIVE together form a programming language which you can use to write programs to manipulate your files. You will find that ARCHIVE programs are simple to write, although the approach is different from writing programs in BASIC. If you have written programs in BASIC before you will see one immediate difference - ARCHIVE programs do not need line numbers. Many of the commands are, however, very like those used in BASIC so you do not have too many new commands to learn. An ARCHIVE program is made up of one or more separate sections. Each section is known as a procedure. A procedure is simply a named section of program. You can then refer to the procedure by its name. In ARCHIVE you can run a procedure by typing its name at the keyboard (and pressing [Enter]). It behaves in the same way as a command - when you write a procedure you are effectively adding a new command to ARCHIVE. Procedures may be as simple or as complex as you want to make them. It is, however, good practice to use lots of short procedures rather than one long one. You will find that you will make fewer programming mistakes. It will also be much easier to find any mistakes that do slip through. In fact there can be up to 255 lines in a procedure, and each line can be up to 255 characters long. A well-written program should never approach either of these limits.

9.1 CREATING A PROCEDURE You must use the program editor whenever you want to write or change a procedure. This editor provides you with many powerful tools for adding, deleting or changing the text of procedures. It is described in detail the next chapter, but in this chapter we shall look briefly at some of the main features so that we can write a few short procedures. We shall assume that initially there are no procedures in the computer's memory. Type:

edit [Enter]

to enter the program editor. You will see that the control area changes to show that you should type in the name for a new procedure. You have been placed directly in the option to create a new procedure. Entering the editor will always lead you to this option if you have not yet defined or loaded any procedures. The first thing to do, therefore, is to define the new procedure. Let us start with a very simple task; to rename the display command. (This, described earlier, shows the contents of the current record in the display area.) The idea is to give the procedure a single letter name and so reduce the amount of typing necessary when using display. We shall give it the name 'd'. Just type in the letter 'd', followed by [Enter]. The sequence of key presses so far, therefore, is:

edit [Enter]
d [Enter]

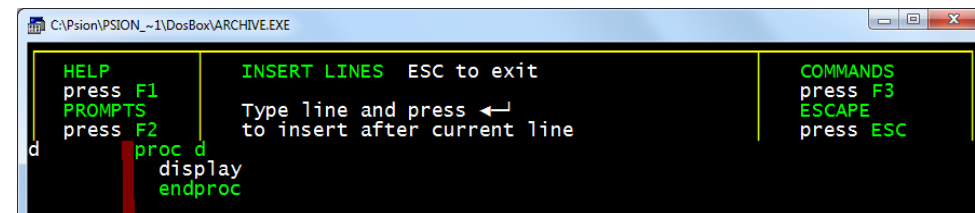
Once you have named the new procedure you will be shown the full range of editing options. Their actions are described in the next chapter. The left hand side of the display area will contain the name of the procedure. The right hand side of the display area will show a listing of the procedure. After the steps described above the screen shows:

```
d
proc d
  endproc
```

You did not need to type in the 'proc' or 'endproc' which mark the beginning and the end of a procedure. ARCHIVE always inserts them automatically when you create a procedure. Once you have given a name to a new procedure, as described above, you have to add the body - that is the sequence of actions that it is to perform. In terms of the current example this means that you must now insert the name of the display command into the procedure. After you have given a name to the new procedure the contents of the control area changes again to show that you can insert lines of text into the new procedure. All you have to do is to type the text of a line, ending it by pressing [Enter]. Type:

display [Enter]

ARCHIVE inserts the new line into the procedure, below the highlighted line. If you have followed this example so far the display will contain:



You could add more lines of text - each line followed by pressing [Enter], would be inserted below the highlighted line. In this case, however, the procedure is complete so you can leave the edit command by pressing [Esc] twice - once to leave line insert and a second time to leave the editor. All you have to do to use the procedure is type its name, followed [Enter]. This new procedure will act in exactly the same way as the display command. Why not try to use this same method to give single-letter names to all the other file display commands; first, last, next, and so on. The next time you use the edit command ARCHIVE will allow you to select from the full set of editing options - remember that you are only directed to the option to create a new procedure when there are no procedures in the computer's memory. You may be puzzled to see that the option to create a new procedure is not one of those shown in the control area. The reason is that it is one of a number of sub-commands within edit. You can select one of these sub-commands by pressing [F3] and then the first letter of the name. To create a new procedure you will have, therefore, to press [F3] and then the N key (for **New** procedure). From this point on the process follows the same method described earlier.

9.2 LISTING AND PRINTING PROCEDURES Whenever you call the edit command you will see that you are shown, at the left of the display area, a list of the names of all the defined procedures present in the computer's memory. You can list any one of these procedures from within the edit command by pressing the [Tab] key (to move down the list) or the [Shift] and [Tab] keys together (to move up the list) until the particular procedure name is highlighted. The procedure is automatically listed at the right hand side of the screen. If the procedure is too long to fit in the display area you will be shown the first part of it. You can then scroll up and down through the procedure with the aid of the up and down cursor keys. When you have finished looking at the procedure listing you can leave the edit command by pressing [Esc]. If you want a printed listing of your procedures you can use the llist command. All you have to do is to type in llist [Enter] and all the procedures currently in the computer's memory will be listed on a printer.

9.3 SAVING AND LOADING PROCEDURES If you want to keep the procedures that you have defined for future use you can do so by using the save command. This stores all defined procedures in a single named file on the disk. If you want to save the new display procedures that you have just defined with a file name "myprocs", you should type in

save "myprocs" [Enter]

At any later time you can bring these procedures back into the computer's memory by typing:

load "myprocs" [Enter]

The load command deletes any existing procedures in memory before loading the new ones from the disk. If you want to add the new procedures to those already in memory, you can do so with the merge command, eg:

merge "myprocs" [Enter]

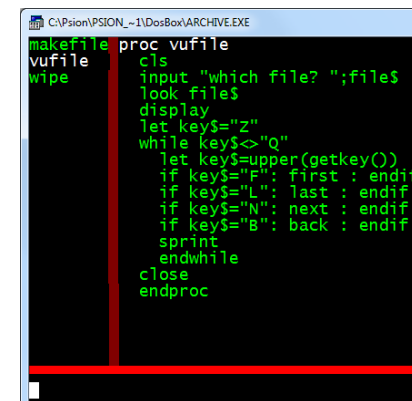
This works like load, except that the existing procedures are not deleted. If a new procedure has the same name as an existing one, the new one will replace the old version.

9.4 EXAMINING FILE RECORDS Renaming commonly-used commands with single-character names by the use of procedures is one way of making life easier for yourself. An alternative way would be to write a longer procedure to replace several commands by single key presses. Try using the edit command to define the following procedure - it allows you to open and examine any of your data files. If you have already defined a procedure, typing:

edit [Enter]

will not automatically give you the option for creating a new procedure. From within the program editor you must press [F3] and then the N key to name a new procedure. Don't worry if you make a few mistakes while typing in the example - you will learn how to correct them in the next chapter. From now on we shall not always show the [Enter] symbol that you must press at the end of each line of input.

```
proc vufile
  cls
  input "which file? ";file$
  look file$
  display
  let key$="Z"
  while key$<>"Q"
    let key$=upper(getkey())
    if key$="F": first :endif
    if key$="L": last :endif
    if key$="N": next :endif
    if key$="B": back :endif
  sprint
  endwhile
  close
endproc
```



Remember that you leave edit by pressing [Esc]. This procedure assumes that there are no existing open data files. Close any open files before using it. Then type: **vufile [Enter]**

It will first clear the display area and then prompt you to type in a file name. When you have entered the name of one of your data files the procedure will open that file in read-only mode and display its first record. It will then wait for you to press a key and will respond to the keys `f`, `l`, `n`, `b` or `q`. The first four of these will cause the appropriate display action (`first`, `last`, `next` or `back`) and pressing the `q` (quit) key will close the file and end the procedure. If you find you have made any typing errors, so that the procedure does not work properly, you can correct them with the aid of the line and program editors described in the next chapter. Since this is the first program of any great length that we have written a few comments might prove helpful. Note how the example is indented to clarify the structure of the procedure. There is no need for you to type it like this, with all the indentations. They are added automatically as you write, list or print the procedure. The main part of the procedure (waiting for a key to be pressed and performing the appropriate action) is enclosed between `while` and `endwhile`. This repetitive loop will only be left when the condition following `while` is false (in this case, when you press the `q` key). The `if` command, used several times within this loop, also requires that each `if` has a matching `endif` to mark the end of the sequence of instructions to be executed if the condition is true. `if` and `endif` are, like `while` and `endwhile`, separate commands and can be used on different lines. We could, for example, have written the first of the `if` statements in this procedure as; `if key$="F" first endif` You may include several lines of statements between `if` and `endif`; they will all be executed, provided the condition following `if` is true. In the `vufile` procedure these statements are sufficiently short that each can be written on a single line, using the colon to separate the individual statements. As you can see, an `sprint` command is used within the main loop of this procedure to make sure that each new record is shown on the screen. Remember that, although the display commands (`first`, `last` etc.) always move to the correct record, the data in the display area is not automatically changed until the end of the procedure. If we had not included the `sprint` command no information would have been shown in the display area until you pressed the `q` key to leave the procedure. In that case all you would see would be the result of the last of any sequence of key presses that you had made. There is more information about procedures and the use of parameters and local variables, in the Programming chapter.

10.0 EDITING PROCEDURES This chapter describes the program editor and how to use it. We shall include a few simple examples, but the best way to learn is by using them yourself. When you have read this chapter you could try writing a few simple programs of your own, or you could try modifying the procedures you typed in while working on the last chapter. If you want to use longer examples you could use the editor to type in all or part of the the programs in the following chapters.

10.1 THE LINE EDITOR Suppose you do not notice a mistake before you press `[Enter]`. ARCHIVE will detect it when it tries to carry out your instructions and give you an error message. Even at this stage all is not lost. You can press

`[F5]` which will put the last line of text you typed in back into the input line. You can then use the line editor to correct the mistake and press `[Enter]` to try again. You can also use the line editor from within the program editor to change a line of one of your procedures. This is described in the following section.

10.2 THE PROGRAM EDITOR You enter the main level of the program editor with the `edit` command by typing:

`edit` `[Enter]`

You can leave `edit` at any time by pressing `[Esc]`. When you enter the program editor the display area changes to show, on the left, a list of the names of any procedures that are in memory. They are always listed in alphabetical order. The first procedure in the list is shown in full on the right hand side of the display area. You will notice that the name of this first procedure is highlighted, as is the first line of its listing. At all stages during the use of the editor highlighting marks the current procedure and the current line within it. This is the line that will be affected by any changes you make. If there are no procedures in the computer's memory at the time you select the `edit` command, the display area will be blank and you are automatically given the opportunity to create a procedure (as described in the previous chapter). Otherwise the control area changes to show the list of the main options available to you. We shall examine each option in turn.

10.2.1 Selecting a Procedure You can select a different current procedure by pressing `[Tab]` to move down the list of procedures, or by pressing `[Shift]` and `[Tab]` together to move up the list. Each time you change the current procedure the listing at the right will change so that it always shows the current procedure.

10.2.2 Selecting a Line You use the up and down cursor keys to select a different current line within the current procedure. The current (selected) line is marked by highlighting. Insertions, will be added immediately after the current line.

10.2.3 Inserting Text You can select the option to insert lines of text below the current line by pressing `[F4]`. Anything you type, up to the next time you press `[Enter]`, is inserted as a new line of text. This new line then becomes the current line. ARCHIVE stays in the insert option so that you can type in several lines; you mark the end of each line by pressing `[Enter]`. When you have finished inserting new text you should leave the option by pressing `[Enter]` twice. You may also leave the insert option by pressing a cursor key or `[Esc]` instead of `[Enter]`. As an example we can create a procedure and add a couple of statements to it. Start with no procedures (type **new**) and then create a new procedure called `test`, using the method described in the previous chapter. Press `[Esc]` twice to leave the editor without adding any statements to the procedure. Then use the `edit` command again to show the procedure. This time, ARCHIVE does not automatically go to the option for naming a new procedure, and the screen shows:

```
test
  proc test
  endproc
```

Press [F4] to use the option to insert lines of text. The highlighting should mark the line including `proc`, so any inserted text will go under this line. Now type:

```
print "this is a test" [Enter]
print "there are two statements" [Enter] [Enter]
```

Pressing [Enter] twice in succession takes you out of the insert option. When you have finished the screen will look like:

```
test
  proc test
  print "this is a test"
  print "it has two statements"
  endproc
```

The highlighting marks the line containing the second print statement. If you make a mistake you can correct it, provided you notice it before you have pressed [Enter], by using the line editor. Remember that you can use this editor at any time that you have typed some text into the input line, before you press [Enter]. Once you have pressed [Enter] the line of text is inserted into the procedure and you will have to use the line editing option, described in the next section, to make any corrections.

10.2.4 Edit a Line From the main level of `edit`, press [F5] to edit the current line. The contents of this line are copied into the input line and you can then edit the text with the line editor. When you press [Enter] ARCHIVE will replace the old line in the procedure with the contents of the input line. You are not allowed to edit the `endproc` statement at the end of the procedure. You are also not allowed to edit the word `proc` in the first line of the procedure, but you may edit the rest of the contents of this line. You can, therefore, rename a procedure by using the line editor to delete the old name and replace it with a new one. The list of procedures at the left of the screen is rearranged automatically to keep the procedures in alphabetical order.

10.2.5 Editing Commands There are four separate editing commands within the `edit` command itself. When you are at the main level of `edit` you can select one of them by pressing [F3] and then typing the first letter of its name. At the end of the action of each of these commands ARCHIVE will go back to the main level of `edit`.

10.2.5.1 Delete Procedure (D) This command deletes the current procedure from your program. You must first select the procedure you want to delete by using the [Shift] and [Tab] keys, as described earlier, to make it the current procedure. You then select the command by pressing [F3] and then the `D` key. You must then press [Enter] to confirm that you really do want to delete the procedure. If you

change your mind at this stage you can, instead of pressing [Enter], press any other key to leave the command and go back to the main level of `edit` without deleting the procedure. Be careful when you use this command since there is no way to restore a deleted procedure, except by typing it in again.

10.2.5.2 New Procedure (N) You will need to use this option whenever you want to start writing a new procedure. As was mentioned earlier you are automatically given this option if you select the `edit` command when there are no procedures in the computer's memory. Otherwise you select it pressing [F3] and then the `N` key. As indicated by the prompt, all you have to do is to type in the name of the procedure you want to create. When you press [Enter] at the end of the name the new procedure becomes the current one, listed at the right of the screen. You are presented with an empty procedure - that is, one containing only the `proc` and `endproc` statements - ready for you to add its body. Note that if you type in the name of an existing procedure the old procedure will be deleted and a new, empty, one will replace it. As an example of creating a new procedure, select the `new procedure` command, by pressing (from within `edit`) [F3] and then `N`, and then type in the name `test` and press [Enter]. You will find that the display area contains:

```
test
  proc test
  endproc
```

ARCHIVE automatically goes straight to the option to insert lines of text into the new procedure - exactly as if you had pressed [F4].

10.2.5.3 Cut (C) This command removes one or more lines of text from the current procedure. The text that is removed can be inserted in another position, or even in another procedure, by means of the `paste` command. Before you select the command you should use the up and down cursor keys to make the current line be either the first or the last line of the section you want to remove. You can then select the command by pressing [F3] and then the `C` key. If you then press [Enter] the current line will be removed from the procedure. Alternatively you can use the up or the down cursor key to move the cursor to the other end of a section of text that you want to remove. The region of text that will be removed is marked by highlighting. When you have marked the text you want to remove you should press [Enter]. ARCHIVE will immediately remove the marked text, placing it in a reserved area of memory known as the paste buffer. The text that is removed replaces any text removed by a previous use of `cut`. If you want to insert the text elsewhere you must therefore use the `paste` command before you use `cut` again.

10.2.5.4 Paste (P) This command inserts the text removed by the last use of the `cut` command into the current procedure, below the current line. The text can be inserted in another position, or even in another procedure. Before you select the command you should, if necessary, use the [Shift] and [Tab] keys to select the procedure in which you want to insert the text. You should also use the up and

down cursor keys to highlight the line immediately above the position where you want to insert the text. You can then select the command by pressing [F3] and then the **P** key. ARCHIVE immediately inserts the text, underneath the current line. When you have used paste to insert the text, the paste buffer is empty. You cannot, therefore, insert the same text in more than one position.

11.0 PROGRAMMING This chapter is about writing programs in the ARCHIVE database language. In addition to explaining the main features of the language, it will describe the development of an actual working example. The example will be developed as we go along, and each new technique will be described as it is needed. Suppose you are involved in running a club or society which charges a subscription and produces a newsletter. You will need to send a copy of each issue to every paid-up member. You will also need to send a reminder to each member when his or her subscription falls due. This example allows you to construct a mailing list and will then print a set of address labels on request. The address label includes a reminder when a subscription is due. The example assumes that you send out six issues of the newsletter per year and that a person's subscription falls due when he or she has received six issues. It could easily be adapted to any situation where you regularly send out some form of circular letter to a number of people on a mailing list.

11.1 A MAILING LIST In this example we shall make as much use as possible of the existing facilities. We can, for example, use the insert and alter commands for all additions and changes to the file records. We shall, however, need to write special routines to print out the address labels. We shall have to cater for the following set of requirements:

- 1) Add a new record to the file
- 2) Delete a record
- 3) Modify a record
- 4) Record subscription payments
- 5) Produce the address labels
- 6) Leave the program

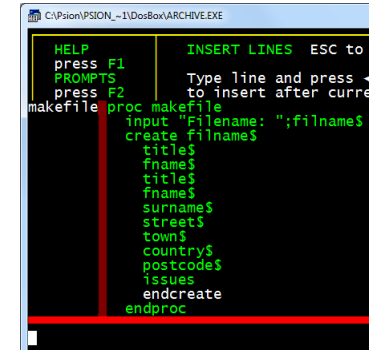
Look for MailList.prg
in the Examples.zip.

```
create "mail"
title$
fname$
surname$
street$
town$
county$
postcode$
issues
(endcreate)
```

We shall write a procedure to handle each of these tasks and link them together by another procedure which will allow you to select any of the options. In this application it is quite clear what fields each record must contain. There will have to be the name and address plus one field to record the number of issues the person has received. We can create the necessary file immediately, as shown below.

We have used three string fields for the person's name; to hold the title (Dr, Mr, Ms etc), the first name and the surname respectively. We could probably have managed with just a single field. There are four string fields for the address, nominally reserved for the street address, the town, county and postcode. You do

not always have to use them in this way, but can treat them as four general fields to hold the address. Four fields should normally be quite sufficient. There is only one numeric field, to hold the information about how many issues have been sent. Note: If you expect to use the same structure for several files you could write a procedure to create the file, using an input command to get the file name. Now that we have the file, we can use it to test the various procedures as we write them. It is a good idea to test each procedure as far as possible as you go along. You can then spot each mistake as it occurs and correct it immediately. If you leave all the testing to the end it will be much more complicated as several things may be going wrong at the same time. Keep things as simple as possible while you are still testing your procedures - try to make sure that each procedure works correctly before you move on to the next one. That way you will find that your final program will usually work as soon as you have written the last procedure.

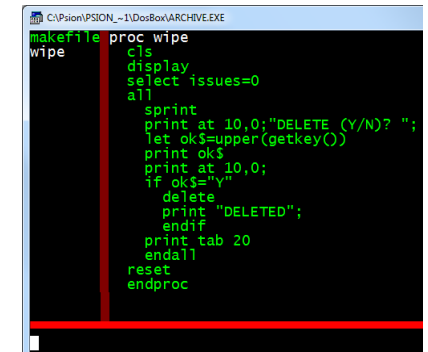


11.1.1 Insertion We do not need to write a procedure to add a record as we can simply use insert. You can use insert immediately to add a few records to the file so that you can test the other procedures on a real file. Remember that you must use sprint to force the display of the contents of the record from within a procedure.

11.1.2 Deletions At some time you will want to remove the records of people who have not renewed their subscriptions. We shall write a procedure, wipe, which allows you to scan through the file, examining the records of all people who have not renewed, and to decide whether each one should be deleted. We shall use the field variable issues to hold the number of issues that a person is entitled to receive. All records for which the value of issues is zero are therefore candidates for deletion.

```

proc wipe
cls
display
select issues =0
all
sprint
print at 10,0;"DELETE (Y/N)? ";
let ok$ =upper(getkey())
print ok$
if ok$="Y"
delete
print "DELETED";
endif
print tab 20
endall
reset
endproc
  
```



```

endif
print tab 20
endall
reset
endproc

```

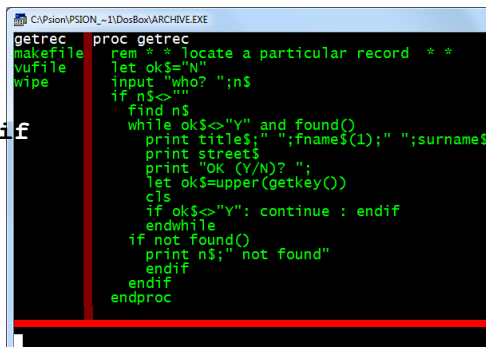
Since a deleted record cannot be recovered, the full contents of the record are displayed and you are asked to confirm that you really want to delete it. We use the `getkey()` function which waits for a key to be pressed and then returns the single-character text string corresponding to that key. Note that `upper()` converts the code to the upper case character so that you can type the letter in either upper or lower case.

11.1.3 Payments You will normally want to record subscription payments from a list of names and addresses of those people who have sent in their subscriptions. You will therefore need to locate the record of a particular person. The best approach is to write a separate procedure, `getrec`, to locate a particular record and then incorporate it in the `pay` procedure. This procedure asks for a text string and then locates the first record in the file which contains that text. If you reply by just pressing [Enter], `n$` is set to the empty string ("") and no search is made. You should use this method to indicate that you have finished recording payments.

```

proc getrec
rem * * locate a particular record * *
let ok$ ="N"
input "who? "; n$
if n$ <> ""
find n$
while ok$ <> "Y" and found()
print title$ ; " "; fname$(1); " "; surname$
print street$
print "OK (Y/N)? ";
let ok$ =upper(getkey())
cls
if ok$ <> "Y": continue: endif
endwhile
if not found()
print n$ ; " not found"
endif
endif
endproc

```



The search uses the `find` command, so that the text is found in any string field. You can therefore identify a record by name or by address. Of course, the first record which matches may not be the one you want, so we have to be able to continue the search. This is the purpose of the `while endwhile` loop. This prints out the name and first line of the address, to identify the record, and asks you if that is the right record. If you do not respond by pressing the Y key, it continues

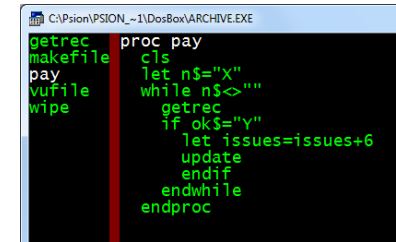
the search. The loop ends either when you answer by pressing the Y key or when the text is not found in any of the remaining records. Note that the function `found()` returns a true (non-zero) value if the search is successful. Since `ok$` could initially be "Y" (from a previous successful search) we must give it some other value at the beginning of the procedure, before entering the loop. This makes sure that the loop will be used at least once. We can now write the `pay` procedure:

```

proc pay
cls
let n$ ="X"
while n$ <> ""
getrec
if ok$ ="Y"
let issues =issues +6
update
endif
endwhile
endproc

```

Look for `MailList.prg` in the `Examples.zip`.



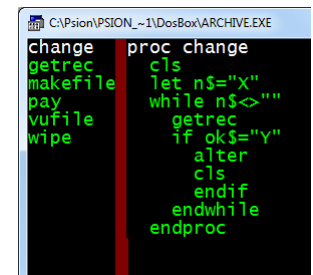
The loop in this procedure continues until `n$` is an empty string. This allows you to record several payments without having to select the `pay` option for each one. When you have finished, just press [Enter] in response to the "who?" prompt. If the value of `ok$` is "Y" after the call to `getrec` then the payment is recorded by marking it as valid for a further six issues. Again we have to set the initial value of `n$` to some appropriate value (anything except "") to make sure that the procedure is not affected by a previous operation.

11.1.4 Changes The procedure to allow you to change the contents of a record is now very easy. Again you must be able to select a particular record to change, so the general structure can be identical to `pay`.

```

proc change
cls
let n$ ="X"
while n$ <> ""
getrec
if ok$ ="Y"
alter
cls
endif
endwhile
endproc

```



11.1.5 Address Labels We use the `lprint` command in the following procedure, to direct the text of the address labels to your printer. When you are writing such a procedure it would be useful to be able to test it by displaying the result on the monitor screen, rather than printing it. Use the `spoolon screen` command for this purpose. After you have typed: `spoolon screen` [Enter] `all lprint` (also `l!ist` and `dump`) output is directed to the monitor screen, rather than the printer. Use

spooloff to cancel a **spoolon** command. We shall assume that the labels are eight lines of print-out in length. If this is not right for your printer and label combination you will have to change the number of blank lines printed by the following procedure, so that it matches your needs.

```
proc dolabel
  if issues
    if issues=1
      lprint "REMINDER - Subscription Now Due"
    else
      lprint
    endif
  lprint
  lprint title$;" ";fname$(1);". ";surname$
  lprint street$
  lprint town$
  lprint county$
  lprint postcode$
  lprint
  let issues = issues-1
  update
endif
endproc
```

Look for MailList.prg in the Examples.zip.

```
C:\Psion\PSION_~1\DosBox\ARCHIVE.EXE
change proc dolabel
dispatch if issues
dolabel if issues=1
getrec lprint "REMINDER - Subscription Now Due"
makefile else
pay lprint
vufile endif
wipe lprint
lprint title$;" ";fname$(1);". ";surname$
lprint street$
lprint town$
lprint county$
lprint postcode$
lprint
let issues = issues-1
update
endif
endproc
```

The procedure includes a reminder in the address label if the person is about to receive his or her last issue. Each time a label is printed, that person's issue count is reduced by one. If this number has reached zero then the label is not printed. Finally we can write the procedure to print all the address labels:

```
proc dispatch
  all
  dolabel
  endall
endproc
```

```
C:\Psion\PSION_~1\DosBox\ARCHIVE.EXE
change proc dispatch
dispatch all
dolabel dolabel
getrec endall
makefile endproc
pay
vufile
wipe
```

11.1.6 Leaving the Program The final option is to leave the program when you have finished. This procedure can be very simple - all it has to do is to make sure that the file is closed properly before returning control to the keyboard interpreter. We have also added a short sign-off message to make it clear that the program has ended.

```
proc bye
  close
  print "bye": stop
endproc
```

```
C:\Psion\PSION_~1\DosBox\ARCHIVE.EXE
change proc bye
bye close
dispatch print "bye": stop
dolabel endproc
getrec
makefile
pay
vufile
wipe
```

Programming ERRORS It is quite likely that sooner or later you will make an error while using this or some other program. You may, for example, accidentally press the [Esc] key or you may type in some text when a number is expected. This type of mistake is detected by ARCHIVE and normally results in the display of an error message and a return from your program to the keyboard interpreter. This could be annoying, to say the least! Fortunately, ARCHIVE has a method by which you can handle all such errors from within the program. You can use the **error** command to mark a procedure to be treated specially if any error is detected. Any error occurring in the marked procedure, or any procedure that it calls, results in an immediate, premature, return from the marked procedure. The normal method of handling errors is switched off for the marked procedure and it is left to you to decide how to deal with it. You can find out the number of the last error that occurred by using the **errnum()** function. You can use it to read the error number more than once as the value is only cleared to zero by the next use of the error command. If no errors have occurred since the start of the program, or since the last time error was executed, then **errnum()** will return a value of zero. This method, although not easy to understand at first, gives you a very powerful and flexible way of dealing with errors. The following example shows a typical way of using error. It gives you an error-resistant method of inputting a number.

```
proc dotest
  input x
endproc

proc test
  let n=1
  while n
    error dotest
    let n = errnum()
    if n print "You made error number " ;n ;", try again"
  endif
endwhile
endproc
```

```
C:\Psion\PSION_~1\DosBox\ARCHIVE.EXE
bye proc dotest
change input x
dispatch endproc
dolabel
dotest
getrec
makefile
pay
test
vufile
wipe
```

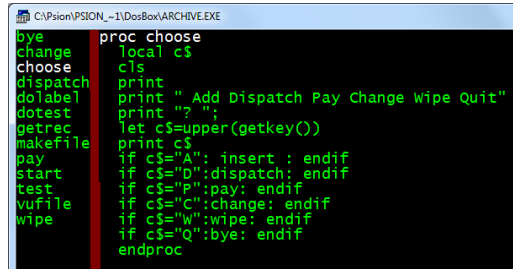
```
C:\Psion\PSION_~1\DosBox\ARCHIVE.EXE
bye proc test
change let n=1
dispatch while n
dolabel error dotest
dotest let n=errnum()
getrec if n print "You made error number ";n; ", try again"
makefile endif
pay endwhile
test endproc
vufile
wipe
```

The first procedure simply waits for your input to the variable x. The second procedure handles any error during the execution of the input procedure. If any error occurs within **dotest** it will be terminated prematurely and the error number will be set. This number is then read by **errnum()** and, if it is non-zero, the error message is printed (this error message could, of course, be anything you like). Since these statements are enclosed in a **while** **endwhile** loop, any error will cause them to be executed again. The error number is cleared by **error**, ready for the next try. You cannot leave **test** until you have typed in a valid number. This example reports the number of the error that was detected. On most occasions you will not be

concerned about which error occurred. The main use of `errnum()` is to differentiate between there being no error - `errnum()` returns zero - and there being a detected error of any type - `errnum()` returns a non-zero value. We can now write a procedure which will allow you to select any one of the six options in the mailing list application with a single key press. The reason for the local command will be explained later. Otherwise the procedure is sufficiently simple that no explanation is necessary.

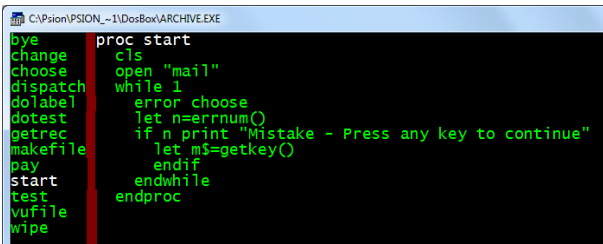
```
proc choose
  local c$
  cls
  print
  print " Add Dispatch Pay Change Wipe Quit"
  print "? ";
  let c$=upper(getkey())
  print c$
  if c$="A": insert : endif
  if c$="D":dispatch: endif
  if c$="P":pay: endif
  if c$="C":change: endif
  if c$="W":wipe: endif
  if c$="Q":bye: endif
endproc
```

Look for MailList.prg in the Examples.zip.



All that remains to be done to complete our program is to write a start-up procedure which opens the file and calls choose. We must include `choose` in a loop so that you are offered the options again, each time you complete your previous selection. You will see that the `while` loop in the following procedure will never end. Such a loop will only come to an end when the expression following `while` has a zero value. In the following procedure the expression (ie 1) is never zero, so the loop will continue indefinitely. The only way of leaving this loop is to choose the **Quit** option. The `stop` command in `bye` immediately returns control to the keyboard interpreter.

```
proc start
  cls
  open "mail"
  while 1
    error choose
    let n=errnum()
    if n print "Mistake - Press any key to continue"
      let m$=getkey()
    endif
  endwhile
endproc
```



Within this loop is a sequence of statements which handles any errors, using a similar method to that described at the beginning of this section. If you make a mistake the program will not continue until you press a key. This allows you to look at what you have just done so that you can find out how you made the error.

11.1.7 THE RUN COMMAND The main procedure in the mailing list program is named `start`. This is so that you can use the run command when using the program. Suppose that, when we have written all the procedures of the program, we save them under the name "maillist". When you want to run the program you will need to load the procedures into the computer's memory and then execute the main procedure, which will call all the others. One way is to use the `load` command and then type in the name of the main procedure, for example:

```
load "maillist" [Enter]
start [Enter]
```

The `run` command will load a named program and then automatically execute the procedure named `start` (if it exists). You can run the program exactly as in the previous example just by typing:

```
run "maillist" [Enter]
```

11.2 PROCEDURE PARAMETERS We shall now describe the use of parameters with procedures. You can use a parameter to pass a value to a procedure, rather than using the value of a variable. Instead of giving a long description of the theory of parameters, we shall show you a few examples of how they can be used. Try the following simple example.

```
proc test;a
  print 5*a
endproc
```

This defines a procedure called `test` which requires one parameter, "a". Notice that the parameter is separated from the name of the procedure by a semicolon. Whenever you use the procedure you must always supply a value for the parameter. For example, you could type:

```
test; 3 [Enter]
```

which will print the value 15 - the number (3) has been passed to the procedure as the value of the parameter `a`. You may specify any number of parameters for a procedure, provided you separate them by commas. For example:

```
proc trial; a,b,c
  print a * b * c
endproc
```

which you can call by: **trial; 3,4,5** [Enter] The values you supply do not have to be literal values, but could be variables, as shown below:

```
let x = 2 [Enter]
let y = 5 [Enter]
let z = 7 [Enter]
trial; x,y,z [Enter]
```

Note that the names of the variables do not have to be the same as the parameter names used within the procedure. We can distinguish between the formal parameters (eg a, b, c) in the definition of the procedure, and the actual parameters which are the actual values that are passed to the procedure. You can also pass the results of expressions:

```
trial; x*2,z/y,(z-y)*x [Enter]
```

You are not restricted to using numeric variables but can also pass strings (or string expressions) as parameters, provided you specify string variables in the definition of the procedure. For example:

```
proc try; a$
  print a$
endproc
```

```
let t$ = "message" [Enter]
try; t$ [Enter]
```

The only requirement is that the number and types of parameters supplied must match the list of formal parameters in the definition of the procedure.

11.3 LOCAL PROCEDURE VARIABLES Most variables that appear in procedures are global. This means that they are recognised throughout the program. They may be used or changed in any procedure, and not just the procedure in which they are first assigned a value. The variables used as formal parameters in a procedure are local variables in that they are not recognised outside of the procedure in which they are defined. The following example may help to clarify the distinction between global and local variables. First we create a procedure which uses two local variables, a and b\$, as well as assigning values to two normal (global) variables, u and v\$.

```
proc demo; a,b$
  print a;b$
  let u = 3
  let v$ = "text"
  print u;v$
endproc
```

Then we use demo : **demo; 5,"words"** [Enter]

All four values are printed, showing that all four variables are recognised inside demo. Typing: print u; v\$ [Enter] shows that both of these variables are also recognised outside the procedure. If, however we try typing:

```
print a; b$ [Enter]
```

we find that they are not recognised outside demo. All formal parameters are local variables, but you can also declare other variables to be local, as in the following example:

```
proc myproc2
  print "inside myproc2"
  print p; q; r
endproc
```

```
proc myproc1
  local q,r
  let p = 2
  let q = 3
  let r = 4
  print "inside myproc1"
  print p; q; r
  myproc2
endproc
```

If you attempt to use myproc1 by typing: **myproc1** [Enter]

you will find that the values of p, q and r are all recognised (and therefore printed) in myproc1, but myproc2 does not know the values of q and r, which are local to myproc1. The values of local variables are not defined anywhere except in the procedure in which they are declared - not even in procedures called from the declaring procedure. The variable p is global and is recognised everywhere. You may be wondering why local variables are necessary. To illustrate their usefulness, suppose you write a program containing several procedures that you, or someone else, originally wrote for use in other programs. It is quite possible that two or more of these procedures might use variables with the same name for quite different purposes. If these variables were global then one procedure could alter a value so that it would be wrong for another. In such a situation you would have to check all the procedures that you use and, if necessary, change the names of the variables. If, however, the variables were local it would not matter if they had the same name. Provided they were in different procedures, changing one would have no effect on the other. Furthermore, it does not matter if a procedure calls another which uses the same name for a variable - provided at least one of them is local. For example, the procedure choose in the section on errors, earlier in this chapter, declared the

variable `c$` to be local. This means that there is no need to check whether any of the many procedures called by `choose` also uses `c$` - the called procedures cannot change the value of `c$` in `choose`.

11.4 Prompts Displaying a prompt and waiting for a key to be pressed is one of the most commonly needed actions, so it is worth writing a general-purpose procedure. The procedure must be able to display a wide range of messages. A simple way of allowing the procedure to print any message is to pass the message to the procedure in the form of a parameter.

```
proc prompt; m$
  print m$+": ";
  let x$=upper(getkey())
  print x$
endproc
```

The message to be displayed is passed to the procedure as a parameter in the local variable `m$`. The function `getkey()` waits for a key to be pressed and returns the ASCII code for the key. In this procedure the ASCII code is converted to upper case by the function `upper()`, so that the result is independent of upper or lower case. Finally the resulting value is assigned to the variable `x$`. This is a global variable, so that the key that was actually pressed is available to any other procedure in the program. A useful procedure is `pause`. It uses `prompt` to print a message and then simply waits until a key is pressed. Since you are not usually interested in knowing which key was actually pressed, it uses a local variable, `y$`, to preserve the original contents of `x$`.

```
proc pause
  rem * * wait for any key * *
  local y$
  let y$ = x$
  print prompt; "press any key to continue"
  let x$ = y$
endproc
```

11.5 DATA ENTRY

11.5.1 Text Accepting text as typed input is quite simple. Any collection of characters is a valid text string (even if it does not make sense) and will not cause a system error. You will not normally need to take any special precautions when accepting text input. It will usually be sufficient to use a line such as the following, which asks you to type in your name:

```
input "Please type your name: ";name$
```

Note that a space is included as the last character of the prompt text; This small point makes a lot of difference to the appearance of your program when you use it. You can input several items with one input statement. All you have to do is to include all the prompts and variable names, separated by semicolons, eg:

```
input "Your first name? ";fname$;" Your surname? ";sname$; [Enter]
```

This last input statement also ends with a semicolon - this stops the cursor moving to the following line after you have typed your input.

11.5.2 Numbers When you use the `input` command to enter text to a string variable the computer will accept anything that you type, without complaint. If, however, you try the same thing with `input` to a numeric variable you will get an error message if you type anything except a valid number or constant numeric expression (eg $3*5+2$). Assuming that you do not want to leave your program every time your finger slips while you are typing in a number, you must make sure that your program can cope with such errors. The most useful way is to make use of the error

command, which was described earlier. The following procedure, for example, will accept any valid number within a specified range. It even produces a prompt message, passed to `getnum` as a parameter.

```
proc getnum; m$,min,max
  local wrong
  let wrong = 1
  while wrong
    print m$; "? ";
    error readnum
    let wrong=errnum()
    if not wrong
      if num<min or num>max
        let wrong=1
        print "Allowed range is ;min; " to ";max
      endif
    endif
    if wrong
      print "Try again"
    endif
  endwhile
endproc
```

Since error must be followed by the name of a procedure, we define `readnum` to input a value for the variable `num`.

```
proc readnum
  input num
endproc
```

Suppose you want to input a value in the range 1 to 10 inclusive, with a prompt message "Numeric value?". You can do this with `getnum`, in the following way:

```
getnum; "Numeric value",1,10 [Enter]
```

11.6 PRODUCING A REPORT Combining `print` (or `lprint` for output to a printer) with an `all` `endall` loop gives an extremely flexible way to generate a report. You can use the `print` items `at` and `tab` to display the report in any format that you want. The first of these moves to a specific line and column position, ready for printing at that point. For example: **print at 5,7; x**

prints the value of the variable `x` at line 5, column 7. Bear in mind that it is quite permissible to use `at` to move upwards or to the left when printing to the screen, but your printer is unlikely to allow you to do this with an `lprint` command. The `tab` item moves the cursor to a specific column within the current line. It does so by printing spaces which, on the screen, will erase any characters in the intervening space. For example: **print tab 25; x**

prints the value of `x` starting at column 25 of the current line. Note that `tab` will have no effect if the `print` position is already at, or beyond, the specified column. Suppose you are considering taking out a full-page advertisement in a magazine. You have a file containing details of a number of possible magazines, including the name of each, its circulation and the cost of a full-page advertisement. The following example shows how you can produce a report for all the magazines, showing the cost per thousand people reached by the advertisement. It assumes you have a file containing the fields `magname$`, `circulation` and `cost`. In order to generate a report with a neatly tabulated appearance we shall start with a procedure to produce centred text in a field of `n` characters. It is useful, for example, in printing titles.

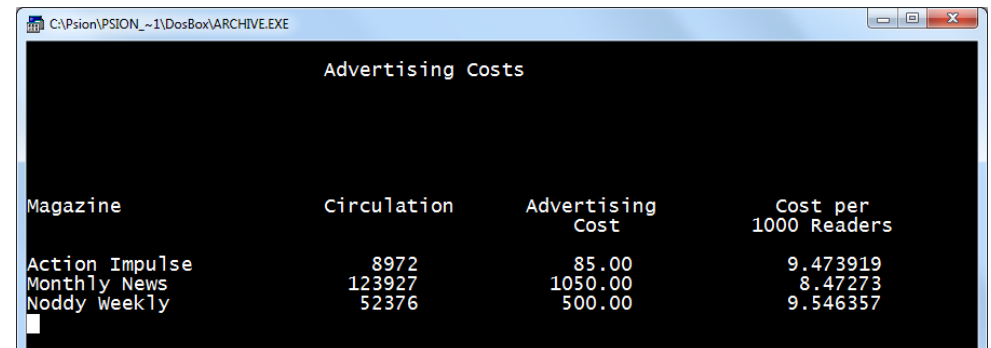
```
proc centre; a$,n
  lprint tab(n-len(a$))/2;a$
endproc
```

Note that the space between the `print` item `tab` and any following text is important. If you do not leave a space, ARCHIVE would attempt to interpret `tab(x)` as a (non-existent) function or a variable. You can print text justified right in a field of `n` characters with the aid of the `dec()`, `gen()` and `num()` functions. Each takes a numeric value and a field width as arguments. The `gen()` and `num()` functions produce the text equivalent of the number in general and integer formats respectively. The `dec()` function produces the text in decimal format. It needs another argument, which specifies the number of figures to appear after the decimal point. If, for example, the variable `x` has the value 27.3 and you type:
let answer\$=dec(x,3,8) lprint answer\$

ARCHIVE will print the text " 27.300" (with two leading spaces).

We can now write the procedure to generate the report, designed for an 80-column display:

```
proc report
  rem * * clear screen and move to line 4 * *
  spoolon screen
  lprint chr(0);chr(12)
  rem * * print title and column headings * *
  centre;"ADVERTISING COSTS",80
  lprint at 8,0;"Magazine"; tab 25;"Circulation";
  lprint tab 42;"Advertising";
  lprint tab 63;"Cost per"
  lprint tab 46;"Cost";
  lprint tab 61;"1000 Readers"
  lprint at 10,0
  rem * * print body of report * *
  all
    lprint magname$;
    lprint tab 25;
    lprint num(circulation,8);
    lprint tab 43;
    lprint dec(cost,2,8);
    lprint tab 64;
    lprint gen(1000*cost/circulation,8)
  endall
endproc
```



For this example we have used the `lprint` command, but have included, as the first line the command: `spoolon screen` to direct printed output to the screen. If you want a printed output, remove this line from the procedure. Note that we have used: `lprint chr(0);chr(12)` When directed to the screen it clears the screen, in a similar way to `cls`. If directed to the printer it causes a form feed, so that the report starts on a new page. Unlike `cls` it does not deactivate any active screen layout.

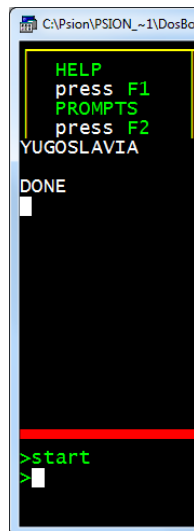
12.0 USING MULTIPLE FILES This chapter extends the explanation of how to use the ARCHIVE programming language by describing how to work with two or more open files. When you have more than one file open at the same time you must be able to identify which file you want to use for any particular operation. You must give each file a unique logical file name when you open or create it and then refer to it by that name in all commands that refer to the file.

12.1 CHANGING A RECORD STRUCTURE Our first example will show you how to add, delete or rename fields within an existing file. Suppose that you want to make some changes to the "gazet" file, to create a new file containing only European countries. In this case the continent\$ field need not be included. The most convenient way of changing the file is to create a second file containing the fields you want and then to copy the required records from the old file to the new one. Let us call the new file "europe". The following procedure will do the rest of the work.

```

proc start
  create "europe" logical "e"
    country$
    capital$
    languages$
    currency$
    population
    gdp
    area
  endcreate
  look "gazet" logical "g"
  select continent$="EUROPE"
  all "g"
    print at 0,0;g.country$;tab 30
    let e.country$=g.country$
    let e.capital$=g.capital$
    let e.languages$=g.languages$
    let e.currency$=g.currency$
    let e.population=g.pop
    let e.gdp=g.gdp
    let e.area=g.area
    append "e"
  endall
  close "e"
  close "g"
  print
  print "DONE"

```



be assumed that the current file is to be used. The last file to be opened automatically becomes the current file. In this example the current file will be "gazet" (with logical file name "g") so we could make use of this by writing the procedure as:

```

proc start
  create "europe" logical "e"
    country$
    capital$
    languages$
    currency$
    population
    gdp
    area
  endcreate
  look "gazet" logical "g"
  select continent$="EUROPE"
  all
    print at 0,0;country$;tab 30
    let e.country$=country$
    let e.capital$=capital$
    let e.languages$=languages$
    let e.currency$=currency$
    let e.population=pop
    let e.gdp=gdp
    let e.area=area
    append "e"
  endall
  close "e"
  close
  print
  print "DONE"
endproc

```

No logical file reference

If you do not include the logical file name in any case where it is optional, ARCHIVE will assume that the command refers to the current file. It is usually safer to include the logical file name explicitly, to avoid any possibility of confusion. You can, at any time, specify the current file by means of the use command. If you included the command: use "e" in the above example, then "europe" would be the current file until you changed it again, either by opening another file or by means of the **use** command.

12.2 THE CURRENT FILE You can see, from the previous example, that you can use the same name for a field in both files - they can be distinguished by including the logical file name. If you do not include the logical file name then it will

12.3 STOCK CONTROL

In a stock control system you will need to:

- 1) Find information on a particular stock item.
- 2) Obtain a report on the current stock levels of all items.
- 3) Record sales and modify the stock records accordingly.
- 4) Order new supplies, to maintain adequate stock levels.
- 5) Record deliveries of stock.

You will obviously need a file to hold the details of all items held in stock and it is convenient to have a second file to hold details of all your suppliers. You will need to be able to access either file from the other - for example you may want to know all the possible suppliers of a particular item, or to find out what items are supplied by a particular company. In order to keep the application as simple as possible we shall not use the menu-driven approach of the examples in the previous chapter. We shall write it as a series of separate commands which can be used - like the standard commands - by typing their names. Since the procedures will be strongly dependent on the file structure we use, we must first give some thought to their appearance.

12.3.1 The Stock File The stock file must contain full details of the stock situation for each item. The following list explains all the fields we shall use.

Field Name	Use	Example
stockno\$	The internal stock code	A101
description\$	Item description	Widget,
large qty	Number in stock	500
sellpr	Selling price	1.25
reorderlev	Reorder when stock level falls below this value	200
buyqty	How many to order	400

We can create the file by:

12.3.2 The Supplier File This file holds the names, addresses and telephone numbers of the companies that supply the goods you sell. It will be useful also to include the name of a contact person in the company. In order to be able to access this information efficiently we shall include a code for each company. We shall use the following fields:

```
create "stock" logical "sto"
  stockno$
  description$
  qty
  reorderlev
  sellpr
  buyqty
endcreate
```

Field Name	Use	Example
coname\$	The company's name	Wonder Widgets plc
street\$	First line of address	36 Acacia Avenue
town\$	Second line of address	Neasden
county\$	Third line of address	London
postcode\$	Last line of address	NW10 1AA
contact\$	Name of a contact	Joe Jensen
tel\$	Telephone number	021-356 1234 ext.212
code\$	Your code for the company	a

We can create the file by:

12.3.3 The Orders File This file effectively forms the link between the previous two files. A typical record would be like below

```
create "supplier" logical
"sup"
  coname$
  street$
  town$
  county$
  postcode$
  contact$
  tel$
  code$
endcreate
```

Field Name	Use	Example
stockno\$	Your stock code	A101
code\$	Your code for the supplier	a
scode\$	The supplier's code for the item	123-456
price	The supplier's selling price	0.98
delivery	The supplier's delivery time (in days)	20

Each record in this file links one record in the stock file with one record in the supplier file. The above example shows that Wonder Widgets (supplier code "a") can supply you with large widgets (stock code "A101"). In addition, we include details of the price, delivery time and the supplier's own stock code. These items are useful when you order more stock.

Using this file allows you to cater for the cases where one supplier supplies more than one stock item (equal values for code\$, but different values for stockno\$) and where one stock item is obtainable from several suppliers (equal stockno\$ but different code\$). Create the file with:

```
create "orders" logical "ord"
  stockno$
  code$
  scode$
  price
  delivery
endcreate
```

12.3.4 Enquiries You will find that the most frequently-needed facility is to find information about a particular stock item, in response to customer enquiries. You will need to find the information as quickly as possible, but may need to find a particular record from either the part number or the description. We shall therefore use the `find` command so that you can give any valid text to start the search. The procedure must be able to ask for you to confirm that the record is the one you want. We shall delegate this task to a separate procedure, as we may want to use it in other situations.

```
proc confirm
  print : print "Confirm (Y/N)";
  let yes=upper(getkey())="Y"
endproc
```

It leaves the variable `yes` containing 1 if you press the Y key - otherwise the value is 0. Note the use of the equals sign for assignment and also in a logical condition (see the description of `search` in Chapter 5).

```
proc inquire
  print
  input "Stock item? "; name$
  use "sto"
  find name$
  let yes=0
  while found() and not yes
  display
  sprint
  confirm
  if not yes
    continue
  endif
  endwhile
  if not found()
  print
  print name$; " does not exist"
  endif
endproc
```

This procedure merely locates the correct record. A more usable procedure for interrogating the stock file is query.

```
proc query
  inquire
  clear
endproc
```

This uses another procedure, `clear` which we shall leave for a moment.

12.3.5 Stock Report We can also write a simple procedure to produce a general stock report, using the method described in the previous chapter.

```
proc report
  cls
  print tab 2; "ITEM"; tab 11; "CODE";
  print tab 20; "QUANTITY"; tab 31; "PRICE";
  print tab 40; "STOCK VALUE"
  print: let total=0: use "sto"
  all
    print description$( to 10); tab 11; stockno$;
    print tab 20; num(qty,6);
    print tab 31; dec(sellpr,2,6);
    print tab 40; dec(sellpr*qty,2,10)
    let total=total+sellpr*qty  endall
  print: print "Total stock value = "; dec(total,2,10)
  clear
endproc
```

12.3.6 Recording Sales All we need to do to record a sale is to subtract the number of items sold from the relevant stock record. It is advisable to include some form of confirmation that we are dealing with the right stock item and that the stock is sufficient to meet the order.

```
proc quantity
  inquire
  if found()
  cls
  input "How many? "; num
  print
  print num;" * ";sto.stockno$;" (";sto.description$;)"
  endif
endproc
```

```
proc sale
  quantity
  if found()
  if num<=sto.qty
  print "Order value: "; num*sto.sellpr
  confirm
  if yes
  let sto.qty=sto.qty-num
  update
  display
  sprint: rem **** show the modified record ****
  endif
  else
```

```

    print "Not enough stock"
  endif
endif
clear
endproc

```

12.3.7 Recording Incoming Stock The following procedure allows you to record the delivery of stock. Again it requests confirmation of the details you type in before accepting them and updating the relevant stock record.

```

proc delivery
  quantity
  if found()
    confirm
    print
    if yes
      print "Accepted"
      let sto.qty=sto.qty+num
      update
      display
      sprint
    else
      print "Delivery not recorded"
    endif
  endif
  clear
endproc

```

12.3.8 Ordering New Stock So far our procedures have only referred to the stock file. When we want to order more stock we shall have to refer to the supplier and orders files for the name and address of the company, the price, and so on. Assuming that we have identified the item in the stock file (with inquire) we select, from the orders file, those records that have the correct stock code. These records contain the codes for all the companies that can supply the item. Since the records also contain the price and delivery time for each supplier, we can decide whether we want the cheapest item or the shortest delivery time. We use locate as a fast way of finding the required supplier record. This means that the supplier file must be ordered (with respect to the supplier code, code\$) before we use doorder.

```

proc doorder
  inquire
  if found()
    use "ord"
    select sto.stockno$=ord.stockno$
    print
    print "fast or cheap (F/C)";

```

```

    if upper(getkey())="F"
      fast
    else : cheap
    endif
    let ycode$=scode$
    reset
    use "sup"
    locate comp$
    doform
      print
      print "Expected delivery is "; del; " days"
    endif
    clear
  endproc

```

The procedure cheap finds the supplier with the lowest price, and fast works in the same way to find the supplier with the shortest delivery time.

```

proc cheap
  use "ord"
  let pri=price
  let comp$=code$
  let del=delivery
  all
  if price<pri
    let pri=price
    let comp$=code$
    let del=delivery
  endif
endall
endproc

```

```

proc fast
  use "ord"
  let del=delivery
  let comp$=code$
  let pri=price
  all
  if delivery<del
    let del=delivery
    let comp$=code$
    let pri=price
  endif
endall
endproc

```

The procedure doform produces the actual order form. You should modify it to your own requirements. We shall use a simple version which shows the order details on the screen.

```

proc doform
  cls
  print
  print sup.coname$
  print sup.street$
  print sup.county$
  print sup.postcode$
  print
  print "Please supply "; sto.buyqty;
  print " * part number ";
  print ycode$

```

```

print "("; sto.description$; ")" ";
print "at "; pri; " each."
print
print "Total value: "; sto.buyqty*pri
endproc

```

The final command that we need is one to close all the files when we have finished using them.

We can now write a short procedure to run the application. It must open all three files with the correct logical file names, clear the display and show you the additional commands that you have. Note that, in normal use, the stock file is the only one whose records will need to be changed. The other two files are opened as read only files. It also orders the supplier file so that we can locate a company by its reference code

```

proc bye
confirm
if yes
cls
print : print "bye"
close "sto"
close "sup"
close "ord"
cls
endif
endproc

```

```

proc start
cls
print at 5,5; "STOCK CONTROL DEMONSTRATION"
print
open "stock"logical "sto"
look "supplier"logical "sup"
look "orders"logical "ord"
use "sup"
order code$; a
clear
endproc

```

An alternative would be to order the supplier file when you create it. Assuming that the supplier file is already ordered, you could then write the above procedure as:

```

proc start
cls
print at 5,5; "STOCK CONTROL DEMONSTRATION"
print
open "stock"logical "sto"
look "supplier"logical "sup"
look "orders"logical "ord"
clear
endproc

```

Finally we can write clear, which simply clears the screen and shows a list of the extra commands available:

```

proc clear
local x$
print
print "Press any key to continue ";
let x$=getkey()
cls
print
print "query report delivery doorder sale bye"
endproc

```

Remember that, in this example, you select a command by typing its full name.

13.0 ARCHIVE REFERENCE

13.1 THE FUNCTION KEYS The function keys are used as follows:

Key	Plus	Use
[F1]		Help
[F2]		Turn the MENU on and off
[F3]		1. show another command menu 2. switch trace on and off, in a running programme 3. select a command, in edit or sedit
[F4]		1. abort, in insert or alter 2. insert text, in edit
[F5]		1. recover last line of input 2. accept record, in insert or alter 3. edit selected line, in edit
[F6]		Freeze the task return to Xchange - <i>Not used in PC FOUR</i>
[F9]		Switch between insert and overwrite
[F10]		Redraw the screen contents
[F4]	CTRL	Move the windows, if using the Pointer Environment with extended resolution - <i>Not used in PC FOUR</i>

13.2 VARIABLES Variable names may be up to thirteen characters in length, and must not start with a digit (0 to 9). They may contain any combination of upper or lower case alphabetic characters, underscore or digits. Other characters are not allowed, except for the dollar sign and the period which have special meanings. If a variable name ends with a dollar sign it is a string variable. Strings may be 0 to 255 characters in length. If the name does not end with a dollar sign the variable is numeric. A variable name may refer to the contents of a record in a file and is then known as a field variable. Field variables are normally assumed to refer to the current file but may be made to refer to another open file by including a logical file name, separated from the variable name by a period. Such a field variable is written as:

logical_filename.fieldname eg main.surname\$

If a variable name includes a dot then it must refer to a field in an open file. If there is no dot an attempt is made to match the name to an existing variable in the following sequence:

- 1) a field of the current file.
- 2) a local variable (a parameter in the current procedure if any)
- 3) a global variable. An error message is given if no match is found.

13.3 EXPRESSIONS An expression is a combination of literal values, variables, functions and operators which results in a single value. A numeric expression results in a numeric value and a string expression results in a text value. Examples are:

3*y*sin(x)+len(a\$) (numeric)
"abc"+a\$+rept("-",5) (string)

An expression may, as in the above examples, be composed of several sub-expressions. In such a case you may not mix sub-expressions of different types. They must all be string expressions or all numeric. A literal string may be enclosed in either double or single quotation marks. Both "text" and 'text' are therefore valid literal strings. The program editor will, however, automatically convert single quotes to double quotation marks if you use them within a procedure. You can include quotation marks in a text string by putting two quotation marks where you want one to appear. For example:

let a\$="abc""def" : print a\$ will print: abc"def

13.4 STRING SLICING You may use a string slicing operation on any string expression. This operation allows you to extract any sequence of one or more characters from a text value. You may add one of the following string slicing operators at the end of any expression that results in a text value.

- (n) select the nth character
- (n to m) select all characters from the nth to mth character inclusive
- (n to) select from character n to end
- (to m) select from the beginning to the mth character

For example, if the ARCHIVE text variable, a\$, has the value "January":

print a\$ (to 3) will print "Jan"
print a\$ (2 to 3) will print "an"
print a\$ (5 to) will print "ary"
let x=2: let y=5: print a\$ (x to y) will print "anua"

13.5 SYNTAX The term syntax refers to the exact structure of a command or function. The syntax of a command, for example, specifies the parameters that the command needs, in what order they must appear, and the symbols (if any) used to separate them. This section describes the notation used to express the syntax of ARCHIVE's programming language.

13.5.1 Syntax Conventions The syntax definitions are:	
Symbol	Meaning
< >	denotes a syntactic entity
[]	encloses an optional item
{ }	encloses items that may be repeated
	the symbol representing "or"
13.5.2 Syntactic Entries	
<s.lit>	literal string
<s.exp>	string expression
<n.exp>	numeric expression
<exp>	expression, either string or numeric
<ptm>	print item
<var>	variable name, either string or numeric
<lfn>	logical file name
<fnm>	physical file name (up to 8 characters)
<pnm>	procedure name

A literal string is text enclosed in quotes, eg "text", or 'text'. A string expression is a literal string, or a combination of literal strings, string variables and string functions that results in a text value, eg "fred"+a\$+chr(72). A numeric expression is either a number, or a combination of numbers, numeric variables and operators (+, -, *, /, etc) that results in a numeric value, eg (3+x)/sin(y). Logical file names and procedure names have the same restrictions as variable names. Physical file names must, in addition, not exceed eight characters. As an example of a syntax definition, consider the syntax of the order command. In our notation it appears as: order <var>;a|d {[,<var>;a|d]} Order therefore needs to be followed by at least one variable name, separated by a semicolon from a letter which must be either a or d. In addition you may optionally include up to 3 further pairs of a variable name and a letter, provided each pair is separated by commas. Clearly, the syntax notation provides a much more compact description. Note that the syntax notation does not tell you the meaning or purpose of the symbols - you will have to read the rest of the description for each command. The syntax only give you a formal description of the kind of items that go to make up a valid command. In addition the syntax notation does not tell you the maximum number of repetitions allowed for the repeated items. Order will accept up to four pairs of a variable and a letter.

13.6 FILE NAMES File names may include a drive identifier and an extension as well as the name of up to eight characters. If you do not supply the drive identifier, ARCHIVE assumes that you are referring to the default data drive (which may be specified with the Xchange command, Set). Keep the directory structure short and without spaces C:\PCFOUR\filename.dbf. You do not normally need to specify an extension since ARCHIVE supplies a default extension for every file access (See section 4.1.5 for default extensions). The look, open and create commands work on database files with an assumed extension of .dbf. The load and save commands supply a default extension of .prg to the program files (unless you include the optional object or protect, in which case they assume an extension of .pro). The default extension for import and export files is .exp, and when you print to a file the default extension is .lis. Screen layout files are loaded and saved by the sload and ssave commands, which assume an extension of .scn. If you include an extension in any file name then it will be used in preference to the default extension normally provided by ARCHIVE. If you start a file name with "_", XCHANGE will use what follows the underscore as the file name, without attempting to add either a default drive or an extension.

13.7 ARCHIVE DATABASE FILES

13.7.1 CONSTRUCTION OF A DATABASE FILE A field is the space reserved to hold either a string or a number. In ARCHIVE, each field is identified by a field variable name, as described in the 13.2 the description of variables. Whether a particular field can hold a string or a number is dependent on the name given to the field at the time it was created - string fields have a name ending with a dollar sign. An ARCHIVE string field may hold up to 255 characters. A numeric field has a name that does not end with a dollar sign. All numbers are stored in the same amount of space, regardless of their value. The possible range for a number is the same as the valid numeric range for the arithmetic operators.

A Record A record is a collection of fields, whose contents are related in some way. The fields of a record might, for example, be used to hold the name, the address and the telephone number of a particular person. In ARCHIVE the records are of variable length so that each record only takes up as much room as is necessary to hold the information contained in its fields. There may be up to 255 fields in an ARCHIVE record.

A Database File A database file is made up from a number of related records. To continue the above example, a database file could consist of a collection of name, address and telephone number records for many different people. The number of records in an ARCHIVE database file is limited to an absolute maximum of about 10900 records, but in practice the limit will usually depend on other factors, such as the size of a record and the amount of storage available on a disk drive. A database file is the basic unit that you can save on, or load from, a disk. Each database file has a name to identify it. In ARCHIVE you give a name to the file when it is created.

13.7.2 Opening and Closing Files When you want to read or write from a database file you must first open it. You can open a database file in read only mode (with `look`) which, as its name suggests, means that you cannot change its contents. However, you can then share the file with other tasks, or even open it more than once in the same task. You also have the option of opening a database file in update mode (with `open`) so that you are allowed both to read and to change its contents. In this case, the file can not be shared with other tasks. Every time you open a database file, ARCHIVE reserves space for the field variables needed by a record of the file. The field variables always contain the values of the current record within the file. When you leave ARCHIVE by means of the quit command, all open files are closed automatically. Do not turn off the computer, or remove a disk from a disk drive, while the disk contains open files as this will leave your database file in a corrupted state. If this does happen, which you will notice by ARCHIVE appearing to do unexpected actions when you perform a database operation, then you must discard the database file and use a previous backup.

13.7.3 Logical File Names Each open database file has an associated logical file name, given to it when the file is opened. If you do not give a logical file name when you open the first file, it is automatically given the logical file name "main". You must supply a logical file name when you open a second or subsequent file. No two open files are allowed to have the same logical name. The logical file name is used to identify a particular file when you are using several files at once. One open file is regarded as the current file. All commands will, in the absence of any other indication, be assumed to refer to this file. It is by default the last file to be opened, but you can make any open file the current file with the `use` command. In addition, many commands allow you to include an optional logical file name to force them to operate on a particular file, even if it is not the current file.

13.8 PROCEDURES A procedure is a named section of program, starting with a procedure declaration of the form:

```
proc <pnm>[ ;<var>{ [,<var>} ]
```

and ending with:

```
endproc
```

It may be referred to by name from any other program or procedure, including itself. It acts as though its code had been inserted at the point from which it is called. In ARCHIVE, the `proc` and `endproc` commands cannot be used directly from the keyboard, but are added automatically when you use the procedure editor to create a procedure.

13.9 PRINT ITEMS A print item is anything, other than a numeric or text expression, that can be included in the list of items in a `print` or an `input`

statement. A print item is one of four possibilities: `at`, `tab`, `ink` or `paper`. A full description of a print item is, in our syntax notation, `<n.exp>, <n.exp> | tab <n.exp> | ink <n.exp> | paper <n.exp>`. You would use `at` to move the cursor to a specific line and column position, ready to print a value. For example, to print the value of the variable `x` starting at line 7, column 12 you could type: `print at 7,12; x` [Enter] Similarly, `tab` moves to a specific column within the current line. It does so by printing spaces until the cursor reaches the specified column. This means that any intervening text will be erased. If the cursor is at or to the right of the specified column, `tab` will have no effect. `ink` and `paper` are print items, as well as being commands in their own right (see the Commands section, later in this chapter). They are also sub-commands in `sedit`. You may use print items in the `lprint` and `input` commands, as well as in `print`.

13.10 THE PROGRAM EDITOR The program editor is entered by means of the `edit` command. If there are no procedures present in memory you will immediately be offered the option of creating a new procedure, as described later. Otherwise you are given a list of all the procedures in memory at the left hand side of the display area. The first procedure is highlighted, and is listed in full on the right of the display. The first line of the procedure is highlighted. This highlighting marks the current procedure and the current line of the procedure. You then have five options which are to:

- 1) Select a procedure.
- 2) Select a line in the current procedure.
- 3) Call an editing command.
- 4) Insert text in the current procedure.
- 5) Edit a line of text in the current procedure.

13.10.1 Select Procedure Press [Tab] to move down the list of procedures. Press [Shift] and [Tab] to move up the list. The listing on the right of the screen always shows the current procedure.

13.10.2 Select Line Press the down cursor key to move to a lower line and the up cursor key to move to an earlier line in the current procedure.

13.10.3 THE EDITING COMMANDS Press [F3] for the menu of editing commands. There are four commands, selected by pressing the key corresponding to the first letter.

Delete procedure - delete the current procedure. Press [Enter] to delete the procedure highlighted on the left of the display. Press any other key to leave the command without deleting the procedure.

New procedure - creates a new procedure. Type in the name of the new procedure and [Enter]. If a procedure of that name already exists you will not

create a new procedure but will be offered the opportunity to edit the named procedure.

Cut - removes text from the current procedure. Text is transferred into the paste buffer. Use the up or down cursor keys to make the first (or last) line of the region to be removed, the current line, before calling this command. Then use the up or down cursor keys to mark the region of text to be removed. Press [Enter] to remove the text into the paste buffer. The new text replaces the old contents of the paste buffer.

Paste - insert text from the paste buffer, below the current line of the current procedure. After inserting the text, the paste buffer is empty.

13.10.4 Inserting Text Press [F4] to select the option of inserting one or more new lines of text below the current line of the current procedure. Then type the line of text and press [Enter]. You can leave this option by pressing [Enter] without any preceding text.

13.10.5 Editing Text Press [F5] to select the option of editing the current line of the current procedure. The line of text is copied into the input line, with the cursor at the start of the editable text in the line. You can then use the line editor to modify the text. Press [Enter] to replace the old line of text with your new text and return to the main level of the edit command.

13.11 THE SCREEN EDITOR The screen editor is entered by means of the `sedit` command. It allows you to design a new screen layout, or to modify an existing one. Once you have designed a layout you can save it on a disk with `ssave` and, at some later time, load it from the disk with `sload`. Think of a screen layout as being composed of two parts - the fixed background text and the variable values that are displayed in it. The `screen` command shows the background on the screen and `sprint` adds the current values of the variables it contains. The main level of `sedit` offers two main options:

- 1) type text into the screen background.
- 2) press [F3] to use a screen editing command.

When typing text into the screen background you can move the cursor around the screen with the four cursor keys and delete existing text. Pressing [Shift] and one of the four cursor keys repeats the last character and moves the cursor in the direction of the cursor key. There are four screen editing commands:

- a) C - clear the screen
- b) V - mark a region to show a variable
- c) I - set the ink colour
- d) P - set the paper colour

A screen layout is made active by:

- 1) `sload`
- 2) `screen`

When a particular screen is active it will show the current values of its variables after a sprint, or when control returns to the keyboard after executing a program (or a command). A screen layout is made inactive by clearing the screen with `cls`. If there is no active screen, `sprint` has no effect. You may only have one screen layout in the computer's memory at any one time. The `display` command creates and uses its own screen layout. It will therefore replace any other screen layout with its own design.

13.12 THE SCREEN DRIVER ARCHIVE controls the screen display by passing control codes to a screen driver program. Screen driver control codes are ASCII values in the range 0 to 31 (decimal) inclusive. You may use the screen driver in ARCHIVE programs, sending the control codes to the screen driver with the `print` command. For example: `print chr(12)` clears the screen and moves the cursor to the top left corner. Some screen driver control codes require one or more additional values, or parameters, to specify their action. Screen driver code 4, for example, requires two additional parameters: `print chr(4)+"S"+chr(10)` This example displays a line of ten S's. Each screen driver code must be followed by the correct number of parameters, otherwise Error 92 (Missing I/O parameter) will result. If the value(s) of the parameter(s) result in an attempt to print outside the limits of the window then Error 93 (Out of range) will be given.

The screen driver codes are listed here, in ASCII order, with full descriptions of their actions. Code 0 No action
Parameters: 0
Code 1 Set ink colour
Parameters: 1
Bits 0 to 2 of the parameter determine the ink colour, according to the this table.

If bit 7 of the parameter is set the current ink colour is saved and the specified colour is set temporarily. Any previously saved ink colour is lost. If bit 6 of the parameter is set the previously saved ink colour is restored. Any colour specified in the parameter is ignored. Example: `print chr(1)+chr(64)` restores the previously saved ink colour
Code 2 Set paper colour
Parameters: 1
Bits 0 to 2 of the parameter determine the paper colour, according to the table as specified for Code 1 Set ink colour. If bit 7 of the parameter is set the current paper colour is saved and the specified colour is set temporarily. Any previously saved paper colour is lost. If bit 6 of the parameter is set the previously saved paper colour is restored. Any colour specified in the parameter is ignored. Example:

Value in bits 0 to 2	40 column	64 and 80 column
0	black	black
1	blue	black
2	red	red
3	magenta	red
4	green	green
5	cyan	green
6	yellow	white
7	white	white

print chr(2)+chr(132) saves the current paper colour the colour is set temporarily to green. Code 3 No action Parameters: 1 Code 4 Repeat characters Parameters: 2 Repeats a single character up to 255 times. The first parameter is the character to be repeated and the second parameter is the number of times to repeat it. Example:

print chr(4)+chr(66)+chr(25) repeats the character B 25 times
 Code 5 Underline Parameters: 0 Switches (or toggles) between underline on/off.
 Code 6 Cursor right Parameters: 0 Moves the cursor one character to the right.
 Code 7 No action Parameters: 0
 Code 8 Cursor left Parameters: 0 Moves the cursor one character to the left.
 Code 9 Tab Parameters: 1 Moves the cursor to the column specified by the parameter, printing spaces along the way. This code has no effect if the cursor is at, or to the right of the specified column. It has a similar effect to the [Tab] print item.
 Code 10 Line feed Parameters: 0 Moves the cursor down one line.
 Code 11 Cursor up Parameters: 0 Moves the cursor up one line.
 Code 12 Form feed Parameters: 0 Clears the screen and homes the cursor to the top left corner.
 Code 13 Carriage return Parameters: 0 Moves the cursor to the left side of the screen window, in the current line.
 Code 14 Cursor on Parameters: 0 Switch on the display of the cursor.
 Code 15 Cursor off Parameters: 0 Switch off the display of the cursor.
 Code 16 No action Parameters: 0
 Code 17 No action Parameters: 0
 Code 18 Character plot type Parameters: 1 Modifies the way in which characters are displayed.

Parameter	Display Effect value
0	ink colour on paper colour
1	ink colour on current background (transparent paper)
2	Exclusive - or character with current display

Code 19 Delete character Parameters: 0 Moves the cursor one space to the left and prints a space. The result is to delete the character to the left of the cursor.
 Code 20 Define window Parameters: 4 Defines the size of a window in absolute screen coordinates. The parameters are, in order, x1 - the left edge (inclusive) y1 - the top edge (inclusive) x2 - the right edge (exclusive) y2 - the bottom edge (exclusive) The window is moved and the cursor is homed to its top left corner.
 Example:

print chr(20)+chr(0)+chr(0)+chr(40)+chr(12) - defines a screen window which is 40 columns wide and 12 rows deep, starting in the top left hand corner.
print chr(20)+chr(0)+chr(0)+chr(80)+chr(24) - defines a full-screen window (80 column mode).

Code 21 Scroll screen up Parameters: 1 Scrolls the window contents up by the number of lines specified by the parameter.
 Code 22 Scroll screen down Parameters: 1 Scrolls the window contents down by the number of lines specified by the parameter.
 Code 23 Scroll left Parameters: 1 Scrolls the window contents left by the number of columns specified by the parameter.
 Code 24 Scroll right Parameters: 1 Scrolls the window contents right by the number of columns specified by the parameter.
 Code 25 Set boundary type Parameters: 1 Determines the behaviour of the cursor at the boundaries of the window. Each bit of the parameter value controls a specific feature.

Bit Boundary	Action if Bit Set	Action if Bit Clear
0	bottom auto scroll up	no auto scroll up
1	top auto scroll down	no auto scroll down
2	right cursor wrap	no cursor wrap
3	left cursor wrap	no cursor wrap
The following behaviours will only occur if cursor wrap (bits 2 and 3) is set at the relevant boundary.		
Bit Boundary	Action if Bit Set	Action if Bit Clear
4 right	toroidal wrap	progressive wrap
5 left	toroidal wrap	progressive wrap

Toroidal wrap leaves the cursor on the same line; progressive wrap moves the cursor to the next line.
 Code 26 Swap ink and paper Parameters: 0 Exchanges the ink and paper colours.
 Code 27 Escape sequences Parameters: 1 Provides a range of miscellaneous extended facilities.

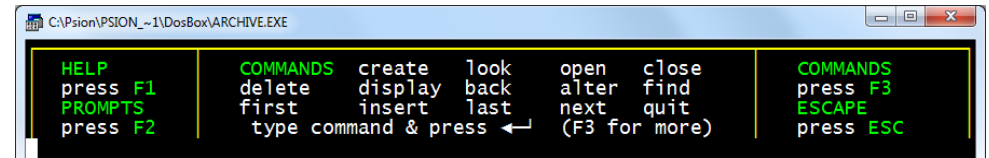
Parameter	Action
chr(65) or "A"	Clears from cursor to end of line
chr(66) or "B"	Clears from cursor to end of window
chr(67) or "C"	Saves current cursor position. previously stored position is lost
chr(68) or "D"	Restores previously stored cursor position
chr(69) or "E"	Scrolls up one line from top of window to cursor.
chr(70) or "F"	Scrolls up one line from cursor to bottom of window.
chr(71) or "G"	Scrolls down one line from top of window to cursor
chr(72) or "H"	Scrolls down one line from cursor to bottom of window

Code 28 CR + LF Parameters: 0 Moves the cursor to the start of the line below. Its action is the same as a carriage return followed by a line feed.

Code 29 Pass through Parameters: 1 The character corresponding to the value of the parameter will be printed on the screen without being interpreted as a screen driver code.

Code 30 Home cursor Parameters: 0 Moves the cursor to the top left of the window. Code 31 Position cursor Parameters: 2 Positions the cursor at the coordinates specified by the parameters. The first parameter is the x-coordinate and the second is the y-coordinate. Its action is similar to the at print item.

13.13 ARCHIVE COMMANDS ARCHIVE commands must be typed in full. There is no need to press [F3] beforehand; this will merely show another menu of commands. The following commands are available:



13.13.1 ALL Syntax: all [<lfm>]:...:endall Scans through all logically present records of the file. The optional logical file name will force all to refer to a specified open file. If the logical file is not given it will scan the current file. The all/endall loop is suitable for situations where you need to examine every (selected) record in a file. Examples include generating reports, finding totals and averages, and calculating maximum or minimum values. The following examples illustrate some of these uses:

```
rem * find total and average of a field named 'age' *
  let total=0
  all
    let total=total+age
  endall
  let average=total/count()
  rem * * * find oldest and youngest ages * * *
  first
  let oldest=age
  let youngest=age
  all
    if age>oldest
      let oldest=age
    endif
    if age<youngest
      let youngest=age
    endif
  endall
```

Do not use **update** inside an all / endall loop, unless you are sure that the length of the record does not change. You can change a numeric field, or you can change the contents of a text field, as long as the length remains the same.

13.13.2 ALTER Syntax: alter Uses the current screen layout to display the current values of the variables. You can change the contents of any of the fields of the current file whose values are shown in the screen layout. Note that it is not necessary for all the field variables to be shown. You cannot change a field that is not shown. If none of the field variables appears in the screen ARCHIVE forces a

display of the file. First select the field to change by pressing [Tab] or [Enter] until the cursor is at the correct field (variables that are not fields of the file are skipped). You can then type a new value or use the line editor to modify the existing value. Press [Tab] or [Enter] to move to the next field. (Pressing [Shift] and [Tab] together moves back to the previous field.) Pressing [Enter] at the end of the last field automatically replaces the old record with the new one. Alternatively you can replace the record at any time by pressing [F5]. If the file is ordered the new version of the record is inserted in sequence. Press [F4] to abort the command - ie to leave alter without the changes you have typed having any effect on the current record.

13.13.3 APPEND Syntax: append [<lfn>] Adds a record to the specified file, or to the current file if the logical file name is not given. The fields of the record take the current values of the field variables. If the file is ordered the insertion is in sequence.

13.13.4 BACK Syntax: back [<lfn>] Moves backwards one record in the specified file, or in the current file if the logical file name is not given.

13.13.5 BACKUP Syntax: backup <oldfilename> as <newfilename> Perform a copy of <oldfilename> into <newfilename>. Use this for making security copies of your database files. backup will automatically compress database files (.dbf) containing redundant (previously deleted) information in the files.

13.13.6 CLOSE Syntax: close [<lfn>] Closes the specified file, or the current file if no logical file name is specified.

13.13.7 CLS Syntax: cls Clears the display area and switches off any display screen. See screen, sload, sprint.

13.13.8 CONTINUE Syntax: continue Continues the previous search or find, from the record following the current record in the current file.

13.13.9 CREATE syntax: create <fnm>[logical <lfn>]:<var>{[:<var>]}:endcreate Creates a named open file whose records contain the fields given by the list of variables specified in the command. You have the option of specifying a logical file name - if you do not, the file is created with the logical file name "main".

13.13.10 DELETE Syntax: delete [<lfn>] Deletes the current record from the specified file, or from the current file if no logical file name is given. Use this command with care since you cannot recover the deleted record.

13.13.11 DIR Syntax: dir [<drive>] Displays a list of files on a disk. ARCHIVE assumes that the drive is the default data drive. You may specify a different drive by typing, for example, D:\.

13.13.12 DISPLAY Syntax: display Shows the logical file name of the current file and a list of the field names and the values of the field variables for the current record. The command replaces any existing user-defined screen layout with this list, which becomes the active screen layout. It is not suitable for showing records containing large numbers of fields.

13.13.13 DUMP Syntax: dump [;<var>]{[:<var>]} Prints the specified fields of the selected records of the current file in tabular form on a printer. You can direct the output to a file or to the screen with spoolon. If you do not give a list of field variable names, all the fields are printed.

13.13.14 EDIT Syntax: edit Calls the procedure editor to create a new procedure or to edit an existing procedure.

13.13.15 ENDALL See ALL.

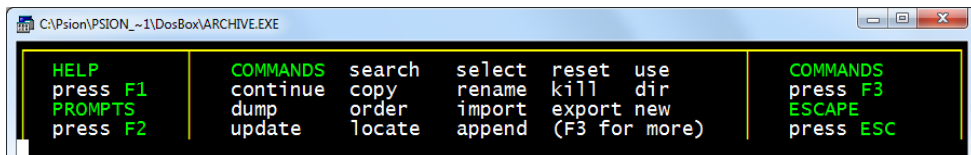
13.13.16 ENDCREATE See CREATE.

13.13.17 ENDWHILE See WHILE.

13.13.18 ERROR Syntax: error <pnm>[:<exp>{[:<exp>]}] Marks a procedure for the purposes of error-handling. Any error which occurs during the execution of this procedure, or any other procedure which it calls, causes a premature return from the marked procedure to the procedure which called it, instead of the whole program aborting. The calling procedure can determine the nature of the error by using the errnum() function to read the error number. This error number is cleared each time that error is executed.

13.13.19 EXPORT Syntax: export <fnm> [;<var>]{[:<var>]}[quill] Saves the named fields of the selected records of the current ARCHIVE file on a disk in a form suitable for import to the other programs in the PC-FOUR family. If you do not specify a list of field variable names, all the fields are exported. If you include the optional parameter quill (separated by at least one space from the last variable name) the file is exported in a form suitable for importing into a Quill document. Do not use the optional quill if you are exporting a file to be used as a secondary file with the mailing list facility in Quill. The export file is named <fnm> and, unless you specify your own file name extension, ARCHIVE uses the extension .exp. See Appendix A for a full discussion of import, export and transfer.

13.13.20 FIND Syntax: find <s.exp> Starts at the beginning of a file and searches for the first record containing a match to the specified string in any string field. The match is independent of upper or lower case text. You can continue the search with the continue command, and determine whether the search was successful by examining the alue returned by the found() function.



13.13.21 FIRST Syntax: first [<lfn>] Sets the file pointer to the first record of the specified file, or the current file if no logical file name is specified.

13.13.22 IF Syntax: if <n.exp> : ... [: else : ...] : endif

- 1) Without the optional else. - If the expression is non-zero the following statements are executed. If the expression is zero execution transfers to the statement following endif.
- 2) With the optional else. - If the numeric expression is non-zero the statements between if and else are executed. Otherwise the statements between else and endif are executed. In either case execution continues with the statements following endif.

13.13.23 IMPORT Syntax: import filename1 as filename2 [logical <lfn>] Reads a file, "filename1", exported from one of the other programs in the PC-FOUR family and produces an ARCHIVE database file "filename2". As with open and look you have the option of specifying a logical file name for the database file. See 'Appendix A' for a full description of import and export file structures.

13.13.24 INK Syntax: ink <n.exp> Sets the foreground colour for all following text to the colour specified by the value of the expression. The colours are: 0 and 1 black 2 and 3 red 4 and 5 green 6 and 7 white If the expression evaluates to more than 7, the value is the remainder after division by 8, ie ink 9 is equivalent to ink 1, both setting the print colour to black. Ink may be used as a print item - ie within a print, or an input command. In this case it will only change the print colour for the duration of the print command. Ink colours have no effect on printed output.

13.13.25 INPUT Syntax: input [<var>|<s.lit>|<ptm>{[;<var>|<s.lit>|<ptm>]}];[:] Requests input from the keyboard to the variables listed in the command. Each variable in an input list may be preceded by an initial string which will be displayed as a prompt for the input. All input items must be separated from each other by semicolons. If the list has a final semicolon the cursor will not move to a new line after the input. The list of input items may include the cursor-positioning items at line, column tab column where line: = <n.exp>, column: = <n.exp> (See the description of Print Items, earlier in this chapter.) You may also use ink and paper as input items. If used within an input command they will only affect the ink and paper colours to the end of the input, when the colours will return to their original settings.

13.13.26 INSERT Syntax: insert Adds a new record to a file. Uses the current screen layout to display the current values of the variables. You can type a new value for any one or more fields of the current file whose values are shown in the screen layout. Note that it is not necessary for all the field variables to be shown. You cannot type a value for a field that is not shown. If none of the field variables appears in the screen ARCHIVE forces a display of the file. First select a field by pressing [Tab] or [Enter] until the cursor is at the correct field (any values on screen that are not fields of the file are skipped). You can then type a new value. Press [Tab] or [Enter] to move to the next field. (Pressing [Shift] and [Tab] together moves back to the previous field.) Pressing [Enter] at the end of the last field automatically adds the new record to the file. Alternatively you can add the record at any time by pressing [F5]. If the file is ordered the new version of the record is inserted in sequence. Press [F4] to leave the command, when you have finished inserting records. You can press [F4] at any time during the use of insert. It leaves insert immediately, without inserting anything you have typed since the last time you inserted a record.

13.13.27 LAST Syntax: last [<lfn>] Sets the file pointer to the last record of the specified file, or the current file if you do not specify a logical file name.

13.13.28 LET Syntax: let <var> = <exp> Assigns a value to a variable.

13.13.29 LLIST Syntax: llist Lists all the procedures currently in memory on a printer. You can use spoolon to divert the listing to a file or to the screen.

13.13.30 LOAD Syntax: load [object] <fnm> Loads the specified program file from a disk into memory. Any procedures previously in memory will be lost. If you include the optional object ARCHIVE will expect the file to be in binary, rather than ASCII, format. (See SAVE)

13.13.31 LOCAL Syntax: local <var>{[,<var>]} Within a procedure, forces the following list of variables to be local variables. These variables exist only within the procedure in which they are declared and are undefined in any other procedure. Their values are destroyed on exit from the procedure.

13.13.32 LOCATE Syntax: locate <exp>{[,<exp>]} Finds - in an ordered file, according to the current index - the first record whose key field(s) match the expression(s). If there is not an exact match locate will still find a record. This record will be the first one whose key fields follow - in the sense of the ordering (ie 'd' follows 'e' if the file is sorted in descending order) - the specified values. This command affects the value returned by the function found(). The function returns a value of 1 only if the located record is an exact match to the specified condition. The record is located much more quickly than if you used find, but the file must first have been sorted. Each expression must explicitly refer to the contents of a particular sort field. In the case of a string field the match is case-dependent, though note that the default collation sequence in PC-FOUR will map lower case

letters to upper case anyway. If you have ordered the file with respect to more than one field you can specify several expressions (one for each sort field). The expressions are separated by commas and must refer to the fields used to order the file. They must be in the same sequence as in the `order` command used to create the current index. For example: `order animal$a,weight;a` locate "Elephant",2000 will find the first record in which the field "animal\$" contains the text "Elephant" and a weight that equals (or exceeds) 2000.

13.13.33 LOOK Syntax: `look <fnm> [logical <lfn>]` Opens the named file for read access only. If the logical file name is not specified it is given the default value "main". Files opened using `look` can be shared with other tasks.

13.13.34 LPRINT Syntax: `lprint [[<exp>|<ptm>]{[:<exp>|<ptm>]}];` Displays the values of the following list of items on a printer, in the same way as for `print`. You can divert the output to a file or to the screen with `spoolon`.

13.13.35 MERGE Syntax: `merge [object] <fnm>` Adds the procedures of the specified program file to the procedures already in the computer's memory. If the file contains a procedure with the same name as one already in memory, the new procedure replaces the old one. If you include the optional object `ARCHIVE` will expect the file to be in binary, rather than ASCII, format. (See `SAVE`)

13.13.36 MODE Syntax: `mode <n.exp> [,<n.exp>]` Changes the form of the display. The first numeric expression may have a value of 0 or 1. A value of 0 joins the control, display and work areas into a single region and turns off the display of prompt messages. A value of 1 separates them back into three distinct areas and restores the display of prompt messages. The second numeric expression may have the values 4,6 or 8, and if used, sets the display to 40, 64 or 80 columns mode. The initial setting is equivalent to: `mode 1`

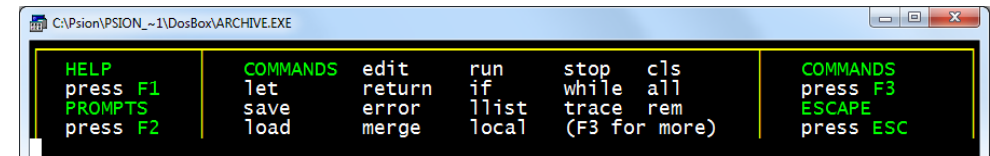
13.13.37 NEW Syntax: `new` Deletes all the data and procedures from the computer's memory, ready for a fresh start. Any open files are closed.

13.13.38 NEXT Syntax: `next [<lfn>]` Moves the file pointer to the next record in the specified file, or in the current file if you do not specify a logical file name.

13.13.39 OPEN Syntax: `open <fnm> [logical <lfn>]` Opens the specified file for both reading and writing. The file is given a logical file name "main" if you do not specify one. Files opened using `open` cannot be shared with other tasks. See `look`.

13.13.40 ORDER Syntax: `order <var>;a[d]{[,<var>;a[d}]}` Orders the records of the current file according to the contents of the specified fields. The first field specified in the list is the primary sort field. Records which have equal contents of their primary sort field are further sorted according to the contents of the next sort field in the list (if it is specified) and so on. For each specified sort field an ordering

direction must be given. This must be either 'a' or 'd' to specify ascending or descending order respectively. The sort fields are combined into a key for each record. Order only takes account of the first 8 characters of a text field. You may not specify more than four fields to be used to construct the key. *The sort order used by Sinclair QL-Archive to order text fields can be configured, using the SuperBASIC program "config.bas".*



13.13.41 PAPER Syntax: `paper <n.exp>` Sets the background colour for all following text to the colour specified by the value of the expression. The colours are: 0 and 1 black 2 and 3 red 4 and 5 green 6 and 7 white. If the expression evaluates to more than 7, the value taken is the remainder after division by 8, ie `paper 11` is equivalent to `paper 3`, both setting the print colour to red. Paper may be used as a print item - ie within a `print` or an `input` command. In this case it will only change the background colour for the duration of that command.

13.13.42 POSITION Syntax: `position <n.exp>` Makes the record whose record number is given by the expression the current record.

13.13.43 PRINT Syntax: `print [[<exp>|<ptm>]{[:<exp>|<ptm>]}];` Displays the values of the following list of items - which must be separated by semicolons - on the screen. If the list has a final semicolon the cursor will not move to a new line after the display. See also `lprint`.

13.13.44 QUIT Syntax: `quit` Closes all files, deletes the current `ARCHIVE` task and closes. **13.13.45 REM** Syntax: `rem` Marks the rest of the line as containing a comment, when used within a procedure. Any following text on that line is ignored when the procedure is executed.

13.13.46 RESET Syntax: `reset` Restores all the records in the current file which were removed by an earlier use of `select` and restores the current index to be the one that was current at the time that `select` was used.

13.13.47 RETURN Syntax: `return` Used within a procedure causes an immediate termination of a procedure by returning to the calling procedure.

13.13.48 RUN Syntax: `run [object] <fnm>` Loads the specified procedure file into memory and starts execution of the procedure called "start". If you include the optional object `ARCHIVE` will expect the file to be in binary, rather than ASCII, format. (See `SAVE`)

13.13.49 SAVE Syntax: save [object][[protect] <fnm> Saves all procedures currently in memory as a single named file on a disk. The program is saved as a file containing ASCII characters. If you include the optional object ARCHIVE will save the file in binary, rather than ASCII, format. This means that ARCHIVE does not have to convert the program into ASCII characters before saving it and is therefore much faster. You can use the load, run and merge commands on such a program by adding the optional object to the appropriate command. These operations will also work more rapidly since no conversion is necessary. Such files have an extension of .pro, rather than the normal .prg. Specifying the protect parameter prevents the object program from being examined on a subsequent load.

13.13.50 SCREEN Syntax: screen Displays the formatted screen layout previously loaded with sload. It does nothing if there is no screen layout present. It does not display any of the variables in the screen.

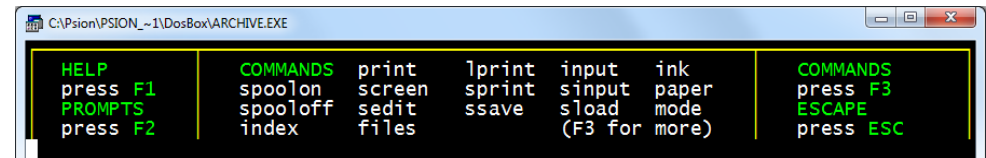
13.13.51 SEARCH Syntax: search <n.exp> Searches the current file from the beginning until a record is found in which the specified expression is true. This record becomes the current record.

13.13.52 SEDIT Calls the screen editor, to enable you to define a new screen layout. See the section on the screen editor, earlier in this chapter.

13.13.53 SELECT Syntax: select <n.exp> Scans the whole file selecting only those records for which the specified expression is true. The file then behaves as if only the selected records are present. You may use repeated select commands to further reduce the number of records present. The selected records are in the same order as in the current index at the time that you used select. If the file was not ordered then the records will be in historical order. You may add, delete or modify the records in a selected file. Note that there is no restriction on the contents of any added or modified records, even if they do not satisfy the condition originally specified in the select command. You can restore all the discarded records with the reset command. Any additions, deletions or modifications that you have made to a selected file are retained after a reset.

13.13.54 SINPUT Syntax: sinput <var>{[,<var>]} Waits for input to the variables in the following list, using the order specified in the list. All the variables in the list must be currently displayed in an active screen layout.

13.13.55 SLOAD Syntax: sload <fnm> Loads a previously defined and saved display screen layout. If called directly from the keyboard it also displays this screen layout and activates the display of any variables within the screen. The displayed values are then updated automatically whenever control returns from a procedure to the keyboard interpreter. If called from a program it does not display the screen layout. In this case you must use screen to display the loaded screen layout.



13.13.56 SPOOLOFF Syntax: spooloff Directs all following lprint and llist output to the printer. This cancels the effect of spoolon.

13.13.57 SPOOLON Syntax: spoolon <fnm> [export | dump] spoolon screen Directs all following lprint, llist and dump output to the specified file - or to the screen - instead of to the printer. If you are directing output to a file it is directed via the currently installed printer driver so that it contains all the special codes that your printer needs. If you include the optional export, ARCHIVE ensures that the file contains only printable ASCII codes, carriage returns and line feeds. The resulting file is suitable for importing into Quill. The optional dump allows the text to be transmitted to the file without being processed by the printer driver. In this case all ASCII codes (including control codes) are passed straight into the file. Unless you specify a file name extension, ARCHIVE assumes an extension of .lis (.exp or .dmp if you include the optional export or dump). The alternative form of the command - spoolon screen directs the output to the monitor screen instead of the printer.

13.13.58 SPRINT Syntax: sprint Forces a display of the fields of the current record. There must be an active screen layout (the screen layout is made active by a previous use of screen, sload or display). If there is no active screen layout the command will have no effect.

13.13.59 SSAVE Syntax: ssave <fnm> Saves, as a named file on a disk, the current display area as a defined screen layout. It saves the text of the screen and a list of the variables in the display, together with their positions.

13.13.60 STOP Syntax: stop Terminates the execution of all procedures and returns control to the keyboard interpreter.

13.13.61 TRACE Syntax: trace [<n.exp>] Controls program tracing when you are debugging a program. When tracing is on, each line of the program is displayed in the work area of the screen, as it is executed. Press the space bar and keep it held down to pause. The trace will continue when you release the space bar. The optional parameter should evaluate to either 0 or 1. If the parameter is 0, tracing is switched off. This is the initial state when ARCHIVE is first entered. If the parameter is 1, tracing is switched on. If the parameter is missing, then tracing is switched (if tracing is off then it is equivalent to trace 1; otherwise it is equivalent to trace 0). You can also switch tracing, but only while a program is running, by pressing [F3].

13.13.62 UPDATE Syntax: update [<lfn>] Replaces the current record in the specified file (or the current file if no logical file name is given) with a record containing the current values of the field variables.

13.13.63 USE Syntax: use <lfn> Makes the specified file the current file.

13.13.64 WHILE Syntax: while <n.exp> : ... : endwhile Repeatedly executes the statements between while and endwhile for as long as the value of the expression is non-zero (true).

13.14 FUNCTIONS Think of a function as a kind of recipe which converts one or more initial values, known as the function's arguments, into a different value, which is said to be the value that is returned by the function. The functions provided by ARCHIVE may take three, two, one or no arguments. The arguments a function are placed in brackets after its name. You must not leave a space between the name and the opening bracket, but spaces are allowed between items within the brackets. If a function takes more than one argument, the arguments are separated by commas. All functions must be followed by the brackets, even if they take no arguments. The presence of the brackets is a useful reminder that you are referring to a function. They allow you to distinguish between a variable and a function, even if they have the same name. The following functions are provided:

13.14.1 ABS (<n.exp>) Returns the absolute value of the argument, ie ignores any minus sign.

13.14.2 ATN (<n.exp>) Returns the angle, in radians, whose tangent is <n.exp>

13.14.3 CHR (<n.exp>) Returns the ASCII character whose code is <n.exp>. A character an ASCII code less than 32 is only sent to the printer if preceded by an ASCII null. For example: lprint chr(0)+chr(12) passes the ASCII character for a form feed to a printer. This is useful if your printer needs control code sequences to produce special effects - refer to your printer manual for any special codes that it needs. You can, for example, send an 'A' to the screen with: print chr(65) For a description of chr() used as a screen driver code, refer to the section on screen driver codes.

13.14.4 CODE (<s.exp>) Returns the ASCII value of the first character found in the specified text.

13.14.5 COS (<n.exp>) Returns the cosine of the given (radian) angle.

13.14.6 COUNT ([<lfn>]) Returns the count of the number of records in the current file.

13.14.7 DATE (<n.exp>) Returns the date as a text string in one of three forms:

<n.exp> date string

0 "YYYY/MM/DD"

1 "DD/MM/YYYY"

2 "MM/DD/YYYY"

You must first have set the system clock.

13.14.8 DAYS (<s.exp>) Returns the number of days, from the first of January 1583, to a date given as a text expression of the form "YYYY/MM/DD". The conversion assumes the Gregorian (modern) calendar, which was first introduced in 1582, is being used and is therefore only valid for dates after 1582.

13.14.9 DEC (value,dp,width) value: =<n.exp> dp: = <n.exp> width: = <n.exp>
Converts the given numeric 'value' to the equivalent text string, in decimal format with 'dp' decimal places. The text is justified right in a field of 'width' characters. For example: dec(1.23e1,3,10) returns the text " 12.300" (with 4 leading spaces).

13.14.10 DEG (<n.exp>) Takes an angle, measured in radians, and converts it to the same angle in degrees.

13.14.11 EOF ([<lfn>]) Returns a value indicating whether you have attempted to read past the end of the current file, or the specified file if a file identifier is given. The value returned is 1 if you have attempted to read past the end of the file, otherwise it is 0.

13.14.12 ERRNUM() Returns the number of the last error which occurred (an error number of zero indicates no errors). The error number is the same as that displayed together with the error message when ARCHIVE reports a detected error.

13.14.13 EXP (<n.exp>) Returns the value of the constant e (approximately 2.718) raised to the power of <n.exp>. The returned value will be in error if <n.exp> is greater than +88, since the result will then exceed the numeric range of ARCHIVE.

13.14.14 FIELDN (<n.exp>[,<lfn>]) Returns the name of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that fieldn(0) returns the name of the first field.

13.14.15 FIELDT(<n.exp>[,<lfn>]) Returns the type of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that fieldt(0) returns the type of the first field. It returns the value 0 if the field is numeric, otherwise it returns 1.

13.14.16 FIELDV(<n.exp>[,<lfn>]) Returns the value of the specified field in the current record of the specified file (or the current file if no logical file name is given). Note that fieldv(0) returns the value of the first field.

13.14.17 FOUND() Returns 1 if a record is found by use of search, find or locate, otherwise it returns 0.

13.14.18 GEN(value,width) value: = <n.exp> width: = <n.exp> Converts the given numeric 'value' to the equivalent text string, in general format. The text is justified right in a field of 'width' characters. For example: gen(1.23e1,10) returns the text " 12.3" (with 6 leading spaces).

13.14.19 GETKEY() Waits for a key to be pressed and returns a single text character which corresponds to the key that was pressed.

13.14.20 INKEY() Returns the single text character corresponding to any key that was being pressed at the time the function is called. It does not wait for a keypress, but will return a null string ("") if no key is pressed.

13.14.21 INSTR(main,sub) main: = <s.exp> sub: = <s.exp> Finds the first occurrence of 'sub' within 'main' and returns the position of the first character of 'sub' in 'main'. It will return a value of zero if no match is found. The match is case-dependent.

i
instr("January","Jan") returns 1
instr("January","an") returns 2
instr("January","AN") returns 0

13.14.22 INT(<n.exp>) Returns the integer value of the number, by truncating at the decimal point. The truncation always operates towards zero. For example:

int(3.7) returns 3
int(-4.8) returns -4

13.14.23 LEN(<s.exp>) Returns the number of characters in the specified text.

13.14.24 LN(<n.exp>) Returns the natural, or base e, logarithm of n. An error results if n is negative or zero, since logarithms are not defined in this range.

13.14.25 LOWER(<s.exp>) Converts the specified text to lower case.

13.14.26 MONTH(<n.exp>) Returns, as text, the name of a month. For example: print month(4) prints the text "April". If an argument larger than 12 is used, it is replaced by the remainder after division by 12 so that, for example, month(13) and month(1) will both give the result "January".

13.14.27 NUM(value,width) value: = <n.exp> width: = <n.exp> Converts the given numeric 'value' to the equivalent text string, in integer format. The text is justified right in a field of 'width' characters. For example: num(1.23e1,10) returns the text " 12" (with 8 leading spaces)

13.14.28 NUMFLD([<lfn>]) Returns the number of fields in the records of the specified file (or the current file if you do not give a logical file name). 13.14.29 PI() Returns the value of the mathematical constant pi.

13.14.30 RAD(<n.exp>) Takes an angle, measured in degrees, and converts it to the same angle in radians.

13.14.31 REPT(<s.exp>,<n.exp>) Returns a string consisting of a number of copies of the first character of the given text. The resulting text may be up to 255 characters in length. For example, print rept("\$",5) prints "\$\$\$\$\$" print rept("abc",3) prints "aaa"

13.14.32 SGN(<n.exp>) Returns +1, -1 or 0, depending on whether the argument is positive, negative or zero.

13.14.33 SIN(<n.exp>) Returns the value of the sine of the specified (radian) angle.

13.14.34 SQR(<n.exp>) Returns the square root of the argument, which must not be negative.

13.14.35 STR(n,type,dp) n: = <n.exp> type: = <n.exp> dp: = <n.exp>
Converts a number, n, to the equivalent text string. The second parameter, type, indicates the form of the converted string as follows;

0 decimal (floating point)
1 exponential, or scientific, notation
2 integer
3 general format

The third parameter, dp, indicates the number of figures after the decimal point in the converted string. It should always be specified, although its value is ignored for integer and general formats. For example:

let a\$=str(12.3456,0,2) gives a\$ the value "12.35"
let a\$=str(12.3456,1,4) gives a\$ the value "1.2346e1"

13.14.36 TAN(<n.exp>) Returns the tangent of the specified (radian) angle.

13.14.37 TASK() Returns, as text, the name of the current ARCHIVE task. You can find out the current task name, for example, by typing: print task() [Enter]

13.14.38 TIME() Returns, as text, the time of day in the format "HH:MM:SS". You must first have set the system clock.

13.14.39 UPPER(<s.exp>) Converts the specified string to upper case.

13.14.40 USR(<n.exp>,<s.exp>) Please refer to section 13.15 for detailed information of the machine code interface in ARCHIVE.

13.14.41 VAL(<s.exp>) Converts the text to its equivalent numeric value. It will only convert text composed of valid numeric characters and the conversion will stop at the first character that can not be interpreted as a digit. For example, val ("1.1ABC") will return the numeric value 1.1, and val ("ABC") will return 0.0.

13.14.42 VALUE(<s.exp>) Returns the value of the variable whose name is given by <s.exp> - for example:

```
let a$="len"
let length=15
print value(a$+"gth") will print the value 15.
```

Note that value(fieldn(y)) is exactly equivalent to fieldv(y).

13.15 ERRORS When ARCHIVE detects an error in a command typed at the keyboard or in a procedure, it displays an error number and a short error message. Examples of errors that would be detected are:

```
attempting to divide by zero
if not matched with an endif
supplying a procedure with the wrong number of parameters
```

If the error comes from keyboard input, the text of the statement remains visible in the work area. You can press **[F5]** to recall the text so that you can use the line editor to correct the error. You can then press [Enter] to execute the corrected statement. If the error comes from a program statement ARCHIVE shows the name of the procedure and the line in which the error occurred. You can then use the program editor to correct the error. When you use the error command in your programs, ARCHIVE will not report any error that it detects in a procedure marked with error. *Nor will it enter the Xchange Disk Full handler if you run out of disk space.* You are free to deal with any such error in any way that you want (including ignoring it). You can find which error has occurred by examining the value returned by `ernum()`. This number is the same as the one ARCHIVE gives when it prints an error message.

13.15.1 ERROR MESSAGES The following list shows ARCHIVE's error numbers, together with the corresponding messages. Where possible, the list includes a short example of a statement that would give the error. The error messages are not designed to pinpoint the precise error, but are intended to give you an idea of what type of error to look for. Those error messages for which there is no short example are marked with an asterisk. They are dealt with in the notes which follow the list.

No	Message	Example
0	(no error)	
1	command not recognized	apend
2	end of statement expected	let x=3 let y=4
3	variable name expected	let 31=x
4	unrecognized print item	print create
5	wrong data type	*(1)
6	numeric expression expected	let x="fred"
7	string expression expected	let x\$=4
8	variable not found	let x=qq (qq undefined)
10	missing separator	print at 5
11	name too long	let thisverylongname=4
12	duplicate name	create"test":n\$:n\$:endcreate (in procedure)
13	string literal expected	*(2)
14	missing endproc	*(3)
15	bad proc statement	*(3)
16	premature end of statement	create"test":endcreate (in procedure)
17	program structure fault	*(4)
18	too many numbers	*(5)
19	key too long	*(6)
20	protected code	load object "myprog":edit
21	file already exists	copy "gazet" as "gazet"
22	too many index files	*(7)
25	too many records	*(8)
50	missing quote	let x\$="fred"
51	missing exponent after 'E'	let x=1.2E
52	number too big	let x=1.2E100
53	unknown symbol	let x=%
70	evaluator syntax error	let x=3+

71	mismatched parenthesis	let x=(3+5)/7)
72	type mismatch	let x\$="fred"+3
73	wrong number of arguments	let x\$=str(1,2)
74	string too long	let x\$=rept("_*",256)
75	divide by zero	let a=0: let x=5/a
76	bad function arguments	let x\$=sqr(-4)
77	string subscript error	let x\$="fred" (3 to 2)
80	out of memory	*(9)
90	no room to open a file	*(10)
91	incomplete file transfer	*(11)
92	missing i/o parameter	*(12)
93	out of range	print at 100,100;37
94	file not open	append (without first opening a file)
100	can not open file	look"xxx" (non-existent)
101	write to read only file	look"gazet":insert
103	wrong file type	sload"gazet" (database file)
104	bad file name	save"3test"
105	error reading file	*(13)
107	disk full	

Notes

*1) The most likely cause of error 5 - 'wrong data type' - is that you input literal text when a number is expected, eg in response to an input statement. Note that the text must be literal, ie enclosed in quotes. It will otherwise be interpreted as a variable name. If the variable does not exist ARCHIVE will give error 8, 'variable not found'. This error message is also given if the two files you specify in the `copy` command with the optional `append` have different record structures.

*2) Error 13 - 'string literal expected' - can occur, for example, during the import of a file that you have constructed yourself (without using any of the export commands in the PC-FOUR programs). It means that ARCHIVE has found a number, or a numeric or text expression, where it was expecting to find a literal text value. In most situations where ARCHIVE finds numeric data when expecting text, or vice versa, it will give error 6 or error 7.

*3) Errors 14 - 'missing endproc' - and 15 - 'bad proc statement' - should never occur in normal use. They indicate that ARCHIVE has detected a missing endproc or an error in the structure of a `proc` statement in a procedure. They are only likely to occur if you construct a program file with an editor other than the one included in ARCHIVE.

*4) Error 17 - 'program structure fault' - usually indicates that an `all`, `if` or `while` is not paired with a corresponding `endall`, `endif` or `endwhile` in a procedure. You can also generate this error by including an `endproc` inside another program structure, or by using `return` directly from the keyboard.

*5) Error 18 - 'too many numbers' - 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.

*6) Error 19 - 'key too long' - means that you are attempting to use an `order` command with more than four fields in the key.

*7) Error 22 - 'too many index files' - means that you are attempting to use `select` or `order` too many times without deleting any of the index files they create.

*8) Error 25 - 'too many records' - ARCHIVE databases have a maximum size of about 10900 records, if unsorted.

*9) Error 80 - 'out of memory' - is very unlikely ever to occur. It may be given if you use a very large program. The size of an ordinary database file is not limited by the amount of memory in the computer since only part of a large file is in memory at any one time. If ARCHIVE ever gives you this error you will have to reduce the size of your program before continuing. You can, if necessary, break your program into several sections, in different files, and chain them with the `run` command. This process will be more efficient if you store the programs as binary files, and load and run them with `run object`. If you have open files when this error occurs, use `new` immediately, to prevent avoid corrupting them.

*10) Error 90 - 'no room to open a file' - occurs when the area of memory Xchange reserves to store internal information about the files currently in memory becomes full. You are not likely to encounter this error unless you work with large numbers of open files.

*11) Error 91 - 'incomplete file transfer' - means that the loading or saving of a file has failed for some reason. This may mean that the data has been corrupted, or that the disk or the disk drive has been damaged.

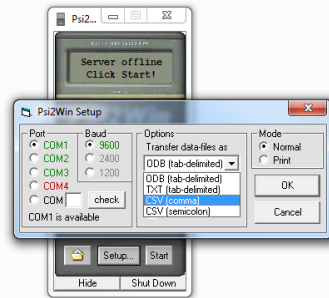
*12) Error 92 - 'missing i/o parameter' - is only likely to occur if you use a screen driver control code with the wrong number of parameters.

*13) Error 105 - 'error reading file' - means that some of the data in a file is in the wrong format, the wrong order, or has been corrupted. This is only likely to occur if you construct your own import file outside PC-FOUR.

14.0 Use Archive to Modify or Create QuizPack Files

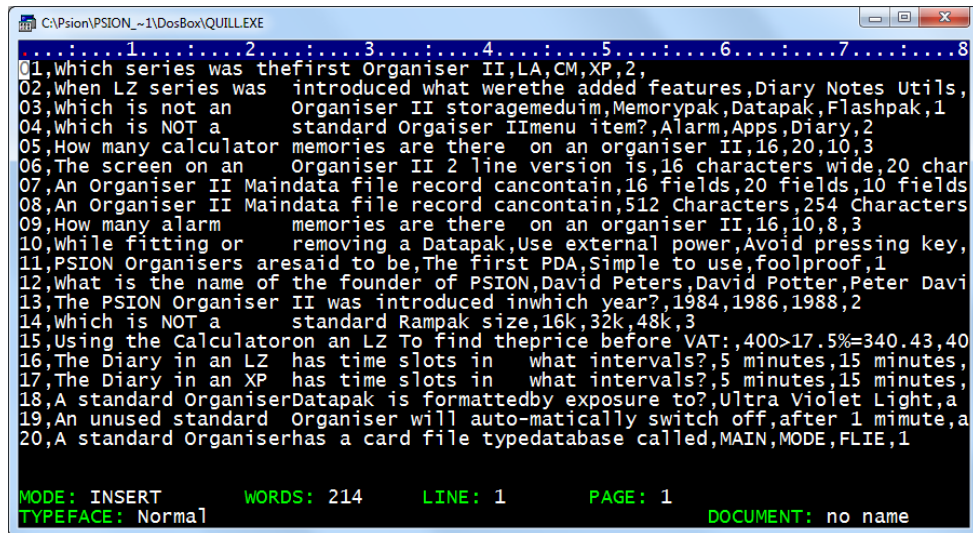
The PSION Organiser II QuizPack programme uses database files to hold the question and answer information. Here we will consider using Archive to modify the QPSION test. You of course could choose any other.

First we should download the existing QPSION database from the organiser to the PC by connecting the Organiser & PC with Psi2Win. Set the Transfer Options to CSV (comma) and use [Comms][Transmit]B:QPSION.



The CSV file will need modifying slightly, For this we can use QUILL. Start QUILL and reset the Margins Left 0, Indent 0, Right 150.

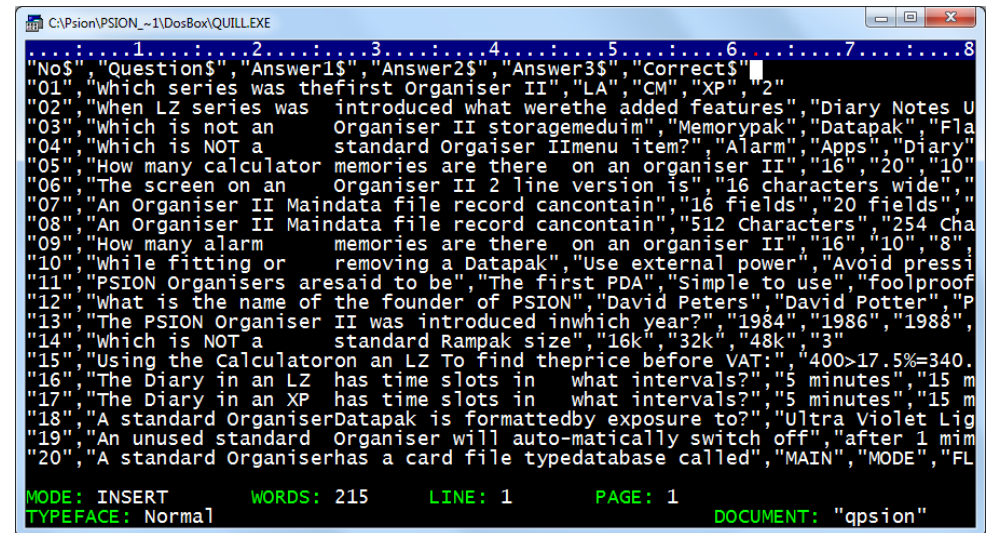
Then we need to import the database into QUILL [F3] Other Files Import QPSION.CSV by line. [F2] turns off prompts



Archive needs the text fields to be enclosed in quotation marks and will need names for the fields. For this exercise we will make all the fields "text" fields.

- Replace the , with “,"
- add “ at the start and end of each record
- add the field names as shown

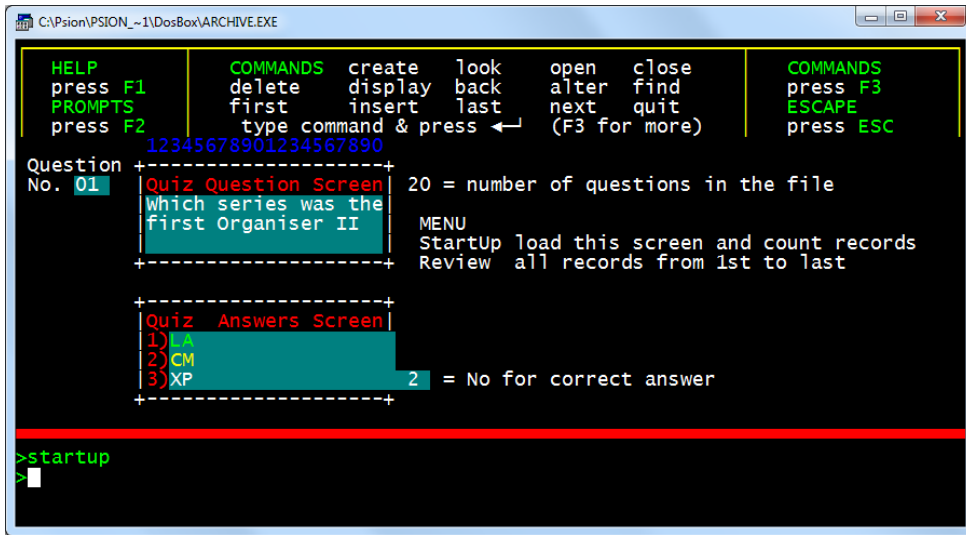
- Put the cursor next to the O1 [F3] Other Replace [Enter] , [Enter] “,” [Enter] [R] 100 times [Space]
- Put the cursor next to the O1 then “ [End] “ [Home] [↓] for each row
- Insert the field names before the first record



Export this file as QPSION.EXP ready of importing into ARCHIVE..



When you have the file imported and displayed in Archive you can then create a data entry screen using SEDIT and SSAYE "QuizPack"



Using a data entry screen like the one here helps format the questions and answers to fit on the PSION Organiser LZ screen. To display the number of records in the file - use something like `max=Count()`. You can set this variable 'max' on the screen with `sedit`.

You can practice writing procedures to create your own MENU of options, two examples are outlined here - Review and StartUp.

When you have the Questions database to your liking then it will need to be prepared ready for sending back to the Organiser.

First.. (for this example)

From Archive... Export "Qarchive.exp"

In Quill [F3] Other Files Import "Qarchive.exp" by line

Then remove the titles from the top line.

Replace all the "," with .

Remove the " from the start and end of each line.

[F3] Other Files Export "Qquill.csv" or some other filename that represents the questions in the file.

All that leaves is to Transmit this CSV file onto the Organiser.

For those interested who do not own a Organiser LZ then all the filesd needed are in the Examples.ZIP.

```
proc Review
  first
  while not eof()
    sprint
    next
    let x$=getkey()
  endwhile
endproc

proc StartUp
  sload "QuizPack"
  let max=count()
  screen
endproc
->
```

Appendix A

Import, Export and Transfer

You can transfer information between the programs of the PC-FOUR family with a comprehensive set of import and export commands. The information used by, for example, ABACUS, ARCHIVE and EASEL is very similar in nature, in that it can always be represented in the form of a table. Transferring information between them is therefore straightforward. QUILL handles formatted text which cannot, in general, be represented in tabular form. It must be treated in a different way from the other members of PC-FOUR. Each member PC-FOUR has its own Export and Import commands, usually as an option within the files command. ARCHIVE, however has separate Export and Import commands within its programming language.

First consider the direct use of the Export command of one task and the Import command of another. These commands are described in the commands section. Export creates a named file which is automatically saved on disk. This file can then be imported to another task, or to several tasks. The export file remains available until you decide to delete it.

Let us first consider export and import between ABACUS, ARCHIVE and EASEL. The export files produced by all three programs are identical in structure and can be imported to any of them, regardless of the program of origin. Suppose we have an ABACUS grid containing the following information ready for Exporting:

	A	B	C	D	E	F	G
1	Cashflow	January	February	March			
2	Sales	1000	1050	1100			
3	Costs	500	530	560			
4	Profits	500	520	540			
5							

If we exported this data and then imported it to EASEL, it would be interpreted as three sets of figures, named 'sales', 'costs' and 'profits'. EASEL interprets the first set of text items that it finds - in this case the month names - as the cell labels for the graph.

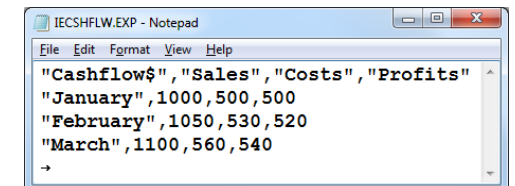
The scheme is:

cell labels>January,February,March

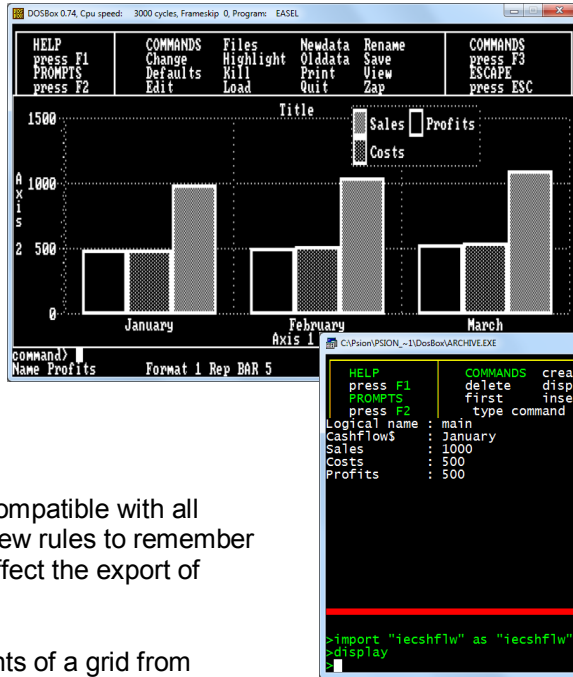
sales graph>1000,1050,1100

costs graph>500,530,560

profits graph>500,520,540



EASEL does not use the first piece of text 'cashflow'. When you export a set of figures from EASEL, it automatically inserts the text 'label' in this position to maintain compatibility with ABACUS and ARCHIVE. If we were to import this same export file to ARCHIVE, the result would be a data file containing three records, each of which would have four field names cashflow\$ (a text field) costs sales and profits (numeric fields).



Rules

To make sure that export files are compatible with all members of PC-FOUR there are a few rules to remember about exported data. They mainly affect the export of information from ABACUS.

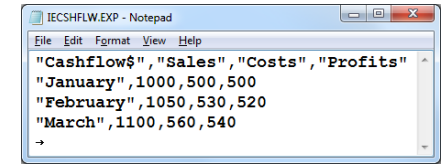
- (1) When you export the contents of a grid from ABACUS, the section of the grid being exported must have text in the first cell of each row (or of each column if you export it in column order). The text must not include spaces but an underscore, for example net_profit is acceptable.
- (2) If the first cell of any row (or column) of an ABACUS grid is empty, that row (or column) is not exported. There must be data in the cell immediately following the text cell in each exported row (or column). The type of this data (numeric or text) determines the data type used for all the data in the rest of that row (or column). Each row (or column) must therefore contain all numeric or all text data.
- (3) You can export files from ABACUS or ARCHIVE which contain several sets of textual data. EASEL can only export a file containing one set of textual data - the cell labels.
- (4) If you import a file containing several sets of data to EASEL, it uses the first set for its cell labels and ignores all following sets of text.

The Export File Structure

The export file consists of a series of records. Each record ends with the two characters <CR> (ASCII code 13) and <LF> (ASCII code 10). The import commands will, however, accept either of these characters - or the two together, in either order - as an end of record marker. The end of the file is marked by a control-Z (^Z) character (ASCII code 26). Each record consists of a series of values, separated by commas. The values are either text (which is enclosed in quotes) or numbers. The first item in each record must be text. If its name ends

with a dollar sign, the following values must be all text. Otherwise the following values must be all numeric. The export file produced by exporting the data of from ABACUS.

```
"cashflow$", "sales", "costs", "profits"<CR><LF>
"January", 1000, 500, 500<CR><LF>
"February", 1050, 530, 520<CR><LF>
"March", 1100, 560, 540<CR><LF>
^Z
```



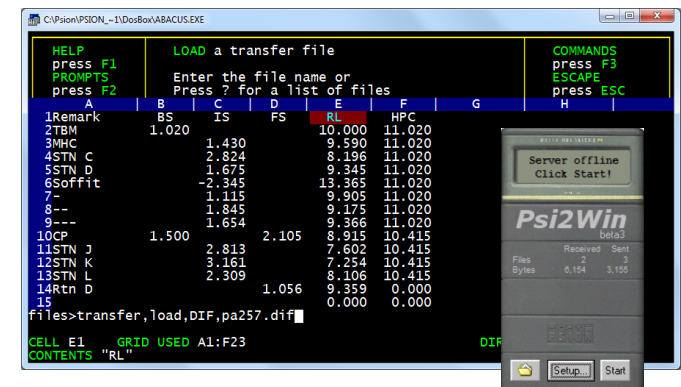
Export and Import for QUILL

Since QUILL works with formatted text, a file imported to QUILL must be a plain text file, rather than the normal tabular export file structure. QUILL will accept any ASCII text containing form feeds and line feeds (ASCII codes 12 and 10 respectively) in addition to the printable ASCII characters. Any other characters in the file are simply ignored. QUILL interprets a line feed as the end of a paragraph and a form feed as a page break. The Export commands of ABACUS and ARCHIVE can produce text files suitable for import by QUILL. ARCHIVE can also export a formatted report to QUILL. You do this by printing (with **Iprint**) the report to a file, using the export option of the spoolon command. A file exported from QUILL contains only plain text and line feed characters, marking the end of each paragraph. In general, a file exported from QUILL is not suitable for import to the other members of PC-FOUR. The main purpose is to be able to produce text files that are readable by a wide range of other programs. (You can, however, write text - containing the necessary quotation marks, dollar signs and commas - that will result in an export file that can be imported to ABACUS, ARCHIVE or EASEL.) One obvious use of export from QUILL is to allow you to write or edit programs. You can, for example, write ARCHIVE programs in QUILL. Once you have exported them, they are in a suitable form for immediate loading and running.

Import from PSION Organiser II Pocket Spreadsheet

Connect the Organiser to a PC with *Psi2Win* select [Plan] and [Load] the pocket spreadsheet file. Use [Mode][File][export][Dif] filename.

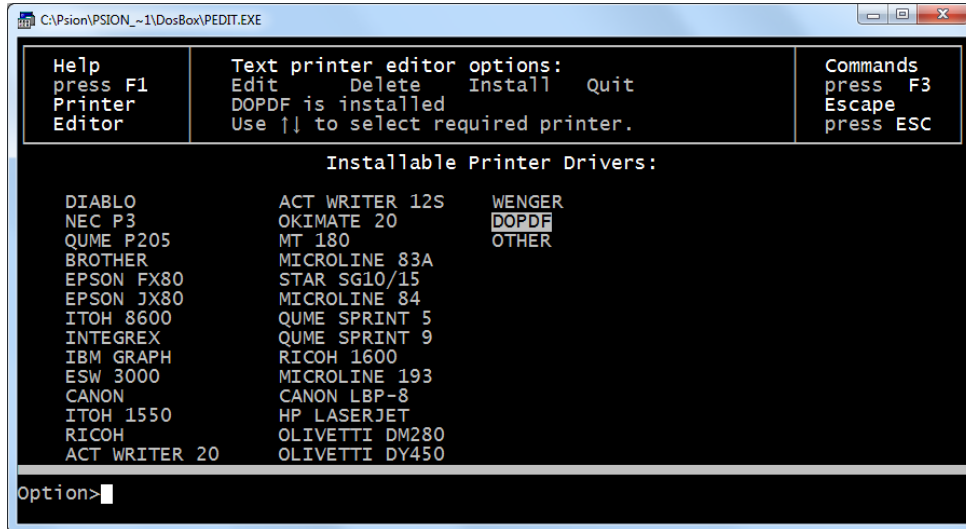
In abacus use [F3] File Transfer Load Diff filename. This will transfer the 'data' but all the formulae and formatting will be lost.



Appendix B

Printer Drivers

Use PEDIT to create an appropriate printer driver. For any PC FOUR application to print it requires a printer configuration file: TPRINT.RES for text based applications and GPRINT.RES for the Easel Application.



To run PEDIT ensure PEDIT.EXE and PRINTER.RES are in the same directory. Use [Edit] to adapt or create a new printer driver. Using [Install] will place (or overwrite) the TPRINT.RES in the same directory.

Appendix C

Zip file contents. PCFOUR.ZIP and Examples.ZIP

PCFOUR.ZIP		PCFOUR.ZIP	
Group.aba	Example Spreadsheet	Unnamed.TMP	Temporary file
GemEasel.app	GEM OS Application	ABACUS.TSL	xchange Task Sequencing Language tutorial files
ABACUS.EXE	PC-FOUR Applications, including the printer driver editor (PEDIT)	ARCHIVE.TSL	
ARCHIVE.EXE		EASEL.TSL	
EASEL.EXE		QUILL.TSL	
PEDIT.EXE			
QUILL.EXE			
Company.dbf	PCFOUR.ZIP Supplied example database files	Examples.ZIP	
Expenses.dbf		Birds.doc	Birds list document
Payrol.dbf		QuillMan.doc	Quill Manual
Persons.dbf		Template.doc	Quill 9.1
Salaries.dbf		Examples.aba	Abacus 3.3
Blunders.exp	Export file for Quill	CashFlow.aba	Abacus 5.1
ABBA.HOB	Application Help Files	BarChart.aba	Abacus 5.2
ARCHV.HOB		Cheques.aba	Abacus 5.4
GRAF.HOB		AutoBar.aba	Abacus 5.6
QUILL.HOB		Mortable.aba	Abacus 5.7.1
Expenses.IX1	Example database index files	MortCalc.aba	Abacus 5.7.2
Payrol.IX1		IRR.aba	Abacus 5.8
Persons.IX1		HPC02.aba	Abacus 5.9
Salaries.IX1		IECshFlw.aba	Abacus Appendix A
Persons.IX2		MailList.prg	Archive 9.0
*.OVL	xchange overlay files	QPSION.csv	Archive 14.0
Demo.prg	Example database programme files	QPSION.exp	
Persons.prg		QuizPack.scn	
Printer.res	Printer driver files	QuizPack.prg	
TPrint.res		Qarchive.exp	
Gazet.scn	Screen layout files for Archive	Qquill.csv	Easel 3.4
Helpp.scr		CashFlow.exp	
Persons.scn		CashFlow.grf	

Manual display suggestions

Print the manual pages - 2 up on A4 landscape single sided. Centre cut to A5 sheets and place in A5 Poly Pocket Ring Binder or A5 Display book presentation folder.

Introduction

The Pison Pocket Spreadsheet can be thought of as a large electronic sheet of paper. This sheet is divided into a grid of boxes, rather like a sheet of graph paper.

This simple arrangement, when transformed by the processing power of the Organiser, forms perhaps the most versatile and powerful tool you can add to your computer.

The information produced on your Pocket Spreadsheet is not just for you alone; it can be shared with colleagues. Worksheets produced on the Organiser can be transferred to a desk-top computer to run in Lotus 1-2-3 or Symphony, and will always produce in Lotus 1-2-3 or Symphony, or any other spreadsheet, what is known as "DF" (default) format. DF worksheets can be transferred just as easily to the Organiser.

The Pocket Spreadsheet gives you the power and flexibility of spreadsheets using many times as much, together with the portability and ease of use of a pocket calculator.

Pison Pocket Spreadsheet does not run on the Pison Organiser Model CM.

Fitting the pack

In the same box as the program pack is a program pack in its original appearance is identical to that of the Organiser, which are identical in appearance. To avoid confusion, the Organiser manual is placed in the pack.

Remove a Desktop or a Pocket Spreadsheet from the original outer box. Push the program pack into the empty slot until it clicks home. The program pack may be used in either of the side slots.


Starting up

The Pison Pocket Spreadsheet requires around 9K of memory, but if large worksheets are to be used then more free memory will be required.

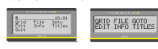
To run the Spreadsheet:

- Press the **[ON/CLEAR]** key to switch on your Organiser.
- If necessary, press the **[ON/CLEAR]** key to make the top-level menu appear.
- Press the **[COPY/SAVE]** key, once more. The Item Plan will have automatically been added to the end of the top-level menu.
- Save Plan from the main menu in the usual way - by moving the cursor to that item with the arrow keys, or by pressing F until it is flashing on Plan, and then pressing **[ENTER]**.

Pison Pocket Spreadsheet Page Menu item at the end of the top-level menu



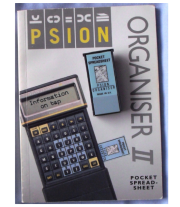
Use **[PAGE]** key at any time to access spreadsheet menu



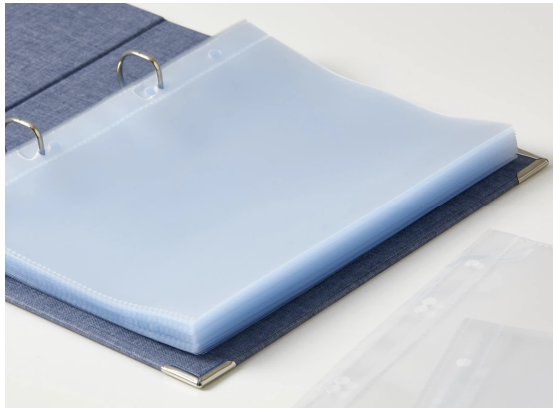
Straight away you will see the top-left corner of the grid. The 'Tutorial' chapters which form Part One of this manual explain where to go from here. However, if you are already familiar with spreadsheets, you can go straight to 'Part Two' the reference chapters, which contain a summary of the particular features of the Pison Pocket Spreadsheet.

Page 1 Page 4

Pison Organiser II - Pocket Spreadsheet



Manual reproduced from archived web information in good faith and not for profit - no liability held - Winton First July 2022 - www.ejbs.co.uk



Pressing **[ON/CLEAR]** when you are editing the range clears your edit and returns to what it was when the command was first activated. In this example:

Delete: **B14:B14**

You can now begin altering the cell range again, as described above, or press **[ON/CLEAR]** to abandon the command and return to the grid.

The commands in the Grid command menu have the following uses:

Delete

Deletes a range of cells from the grid. When Delete is selected the Spreadsheet prompts for a range of cells.

When the cells in the range indicated have had their contents deleted, there is no way to retrieve them. Use this command with care.

Format

Sets the numeric display format for either a specified range of cells or for all blank cells in the grid.

When Format is selected, three item menus are displayed, containing the items Default and Cells.

- When Default is selected, no cells already containing data will be affected.
- When Cells is chosen, the Spreadsheet prompts for the cell range in which the command is to operate.

Once the cell range has been entered if you select Cells, it immediately if you select Default. If a last item menu is displayed:

When Fixed or Scientific are selected, the bottom line of the display prompts for the number of decimal places. In both cases, the default 2 is offered and the maximum allowed is 11.

- Fixed** displays a number to a fixed number of decimal places.
- Scientific** presents a number in the format 3.12E+2 where 2 is the number of decimal places you select (E2 is 10, E3 100 etc).

When Range or General are selected, the display returns immediately to the grid with that number; display format is affected.

- Range** format is the same as Fixed format with 11 decimal places.
- General** represents numbers in the best format seen in this order: Integer, Fixed, Scientific.

Page 47

Only when contents may be 1, 2 and 3 are 1 to 11 characters that width. The width (column) is to affect, the column width. The maximum for column A) The 2 and 3 characters or mode 4 page column of a line, i.e. a 110 order.

See Pison Pocket Spreadsheet 3004 page 4 option for print directly to PDP or Printer via USB Camera link.

the grid is not the size of the grid. rectangular into underlying area are printed. grid for a page.

Page 48

The Print command will operate only when a Comm Link cable or Pison Printer is connected to the Organiser. If neither of these is connected, a DEVICE MISSING error will be displayed. You will then be returned to the grid.

Page 49