

This is Advanced Pick

Malcolm Bull

Malcolm Bull

Training & Consultancy Publications

© Malcolm Bull 1994

Malcolm Bull Training & Consultancy Publications

19 Smith House Lane

BRIGHOUSE

HD6 2JY

West Yorkshire

United Kingdom

Telephone: +44-01484-713577

Fax: +44-01484-713577

ISBN: 1 873283 57 1

Edition 1: August 1994

Reprinted with corrections: December 1994

2: January 1995 (US Edition)

Whilst every care has been taken in the production of the book and all associated materials, Malcolm Bull can assume no liability with respect to the document nor to the use of the information presented therein.

The *Pick system*, *Open Architecture* and *Advanced Pick* are proprietary software products of Pick Systems, Irvine, California, USA. This publication contains material whose use is restricted to authorised users of the Pick system. Any other use of the descriptions and information contained herein is improper.

The use in this book of the names *PICK*, *ADVANCED PICK*, *OPEN ARCHITECTURE*, *ADDS MENTOR*, *C.ITOH*, *FUJITSU*, *McDONNELL DOUGLAS*, *PRIME INFORMATION*, *REALITY*, *REVELATION* and *ADVANCED REVELATION*, *SANYO*, *UNIDATA*, *UNIVERSE*, and all other trademarks and registered marks is gratefully acknowledged and respected.

Contents

Preface	vii
Acknowledgements	x
Conventions	xi
1 Introduction to Advanced Pick	1-1
1.1 Advanced Pick	1-3
1.2 Implementations of Advanced Pick	1-4
2 Moving to Advanced Pick	2-1
2.1 Differences between R83 Pick and Advanced Pick	2-1
2.2 Converting from R83 to AP	2-23
3 System architecture	3-1
3.1 Ports	3-3
3.2 Printers	3-3
3.3 Hard disk	3-4
3.4 Backing storage	3-11
4 File structure	4-1
4.1 The physical organisation	4-2
4.2 The structure of files	4-10
4.3 Accounts	4-15
4.4 The hierarchy of files	4-16
4.5 Item format	4-19
4.6 Working with files	4-22
4.7 Standard files	4-37
4.8 Working with items	4-43
5 File design	5-1
5.1 File names	5-2
5.2 Modulo and file efficiency	5-2
5.3 Choosing a modulo	5-4
5.4 Dictionary design	5-6
5.5 Item-id design	5-8
5.6 Data design	5-9
5.7 Security aspects	5-12
6 Using your Advanced Pick system	6-1
6.1 Switching on	6-1
6.2 Logging on	6-2
6.3 When you have logged on	6-2

6.4	Logging off	6-5
6.5	Switching off	6-6
6.6	Using your application system	6-6
7	The Advanced Pick keyboard	7-1
7.1	<Ctrl> S	7-3
7.2	Redefining the user keyboard	7-3
7.3	SET-KBRD	7-5
7.4	SET-FUNC	7-6
8	TCL and TCL commands	8-1
8.1	General notes on TCL	8-3
8.2	TCL options	8-4
8.3	Case-sensitivity	8-5
8.4	TIMEDATE	8-7
8.5	Stacker	8-8
8.6	Capturing TCL history	8-10
8.7	Level pushing	8-10
8.8	LOGTO and level pushing	8-13
8.9	Macros	8-13
8.10	Menus	8-17
8.11	Important TCL commands	8-19
8.12	Phantom processes	8-22
8.13	GraPICK: the Pick GUI	8-23
9	The Editor	9-1
9.1	General points about the Editor	9-2
9.2	EDIT command	9-3
9.3	Editor commands	9-4
9.4	Prestore commands	9-11
9.5	RECOVER-FD	9-14
10	The Update Processor	10-1
10.1	UPDATE command	10-1
10.2	Some examples of the UPDATE command	10-2
10.3	DATA-ENTRY	10-4
10.4	ID-PROMPT	10-4
10.5	The UD command	10-5
10.6	D-pointers and the Update Processor	10-6
10.7	Attribute definitions and the Update Processor	10-7
10.8	UP versus application programs	10-8
10.9	Using the Update Processor	10-9
10.10	The Update Processor and Basic programs	10-10
10.11	Living with the Update Processor	10-11
10.12	Special UP facilities	10-14
10.13	Processing codes used with the UP	10-30

11	The Output Processor & Runoff	11-1
11.1	Creating and printing an OP document	11-2
11.2	OP command	11-5
11.3	Output Processor commands	11-5
11.4	Quick OP	11-6
11.5	Runoff	11-8
11.6	Creating and printing a Runoff document	11-9
11.7	Why Runoff?	11-10
11.8	RUNOFF command	11-10
11.9	Quick Runoff	11-10
12	Access : the enquiry language	12-1
12.1	Introducing Access	12-1
12.2	Format of Access sentences	12-2
12.3	Access verbs	12-4
12.4	The file name	12-6
12.5	Output list	12-6
12.6	Non-columnar reports	12-7
12.7	SAMPLING: selecting the items	12-13
12.8	Sort specifications	12-19
12.9	BREAK-ON and control breaks	12-21
12.10	TOTAL – totals and subtotals	12-24
12.11	GRAND-TOTAL	12-25
12.12	LPTR and printed reports	12-26
12.13	HEADINGS and FOOTINGS	12-28
12.14	Other modifiers and connectives	12-30
12.15	Predefined phrases	12-32
12.16	Select-lists and saved-lists	12-34
12.17	Access options	12-40
12.18	LIST-LABEL / SORT-LABEL	12-41
12.19	REFORMAT / SREFORMAT	12-43
12.20	Access and SQL	12-45
13	Access & attribute definitions	13-1
13.1	Format of attribute definitions	13-1
13.2	Conversion or correlative?	13-8
13.3	Creating an attribute definition	13-9
13.4	Examples	13-10
14	Access processing codes	14-1
14.1	A : Arithmetic and logical processing	14-1
14.2	B : Update Processor	14-6
14.3	C : Concatenation	14-7
14.4	CALL : Calling a Basic subroutine	14-8
14.5	CU : Update Processor	14-10
14.6	D : Date conversion	14-10
14.7	F : Arithmetic and logical processing	14-13

14.8	G : Group extraction	14-14
14.9	I : File indexing / Update Processor	14-15
14.10	ID : Update Processor	14-16
14.11	IF : Logical processing	14-18
14.12	L : Length	14-19
14.13	MC : Mask characters	14-19
14.14	MD / ML / MR : Mask decimal values	14-21
14.15	MT : Mask time values	14-22
14.16	P : Pattern matching	14-23
14.17	R : Range checking	14-24
14.18	S : Substitute zero and null values	14-24
14.19	T : Text substring extraction	14-25
14.20	Tfile : File translation	14-25
14.21	Xc : Update Processor	14-28
14.22	Other codes	14-28
15	File indexing	15-1
15.1	Creating the index	15-1
15.2	Using the index in Access	15-4
15.3	Using the index in Basic programs	15-4
15.4	The B-tree index	15-5
16	The Basic language	16-1
16.1	Basic programs	16-2
16.2	Variables, literals, numbers and strings	16-8
16.3	Dimensioned arrays	16-12
16.4	Dynamic arrays	16-16
16.5	Program structures	16-23
16.6	Calculations and assignment statements	16-34
16.7	Displayed and printed output	16-45
16.8	Keyboard input	16-53
16.9	Subroutines	16-58
16.10	INCLUDE statement	16-66
16.11	EXECUTE statement	16-66
16.12	TCLREAD statement / ERROR() function	16-70
16.13	File handling	16-72
16.14	Basic and backing storage	16-83
16.15	TCL commands for Basic programs	16-84
16.16	Flash/Basic	16-91
17	Procs	17-1
17.1	Simple Procs	17-3
17.2	Buffers and stacks	17-8
17.3	Handling the buffers	17-10
17.4	P statements	17-12
17.5	Output: Displaying information	17-12
17.6	Control statements: GO / IF / X	17-14

17.7	Subroutines	17-16
17.8	Arithmetic in Procs	17-18
17.9	File handling	17-18
18	The debugging software	18-1
18.1	Escape or debug?	18-1
18.2	Fundamental responses	18-2
18.3	The Basic debugger	18-4
18.4	The system debugger	18-13
19	Printing : the spooler	19-1
19.1	The users, the form-queues and the printers	19-2
19.2	When do you use the spooler?	19-4
19.3	Starting the spooler	19-5
19.4	SP-ASSIGN	19-8
19.5	SP-KILL	19-10
19.6	LISTPEQS	19-11
19.7	SP-EDIT	19-12
19.8	AP/Unix and the spooler	19-16
20	Backing storage	20-1
20.1	Concepts	20-1
20.2	Choosing the backing storage device	20-3
20.3	TCL commands for handling backing storage	20-6
20.4	Using backing storage	20-9
20.5	Access and backing storage devices	20-9
21	Operations : Saving & restoring data	21-1
21.1	Saving data	21-2
21.2	File statistics	21-9
21.3	The SAVE verb	21-10
21.4	Verifying the save	21-11
21.5	Restoring data	21-12
21.6	Transaction logging	21-18
21.7	File resizing	21-19
22	Operations : System administration	22-1
22.1	Starting and stopping the system	22-1
22.2	Users and accounts	22-2
22.3	Details of the system	22-8
22.4	The date and the time	22-13
22.5	Messages	22-14
22.6	Accounting statistics	22-16
22.7	Terminal characteristics	22-17
22.8	Some further topics	22-20
23	Security	23-1

23.1	Logging on and passwords	23-1
23.2	System privileges	23-2
23.3	Lock-codes	23-3
23.4	Logon procedures	23-5
23.5	Closed systems	23-6
23.6	Other security features	23-7
24	Group format errors : GFEs	24-1
24.1	Nature of GFEs	24-2
24.2	Causes of GFEs	24-3
24.3	Locating GFEs	24-5
24.4	Fixing GFEs	24-6
24.5	GFEs and file-saves	24-7
24.6	Recovering from GFEs	24-7
25	AP & the host operating system	25-1
25.1	Shelling out	25-1
25.2	Exporting to AP/DOS	25-2
25.3	Exporting to AP/Unix	25-2
25.4	Exporting from AP/Native to diskette/tape	25-3
25.5	Importing from AP/DOS	25-3
25.6	Importing from AP/Unix	25-4
25.7	Importing to AP/Native from diskette/tape	25-4

Appendix:

A	Glossary	Appendix-1
B	Output processor commands	Appendix-8
C	Runoff commands	Appendix-9
D	Heading / footing options	Appendix-10
E	Basic statements	Appendix-12
F	Basic functions	Appendix-13
G	Proc statements	Appendix-14
H	ASCII characters	Appendix-15
I	Advanced Pick <Ctrl> keys	Appendix-17

Index

Preface

For over two decades, the Pick system has been one of the leading computer environments for commercial users. Since 1984, there have been around 180,000 installations of the R83 Pick system in over 35 countries around the world. Since it was launched in 1988, there have been around 150,000 installations of Advanced Pick, and with European sales increasing at a rate of 18% per year and world-wide sales increasing at a rate of 15% per year, the Pick system is surely the foremost commercial computing system in use today and it is likely that the Pick system – which has been described as **the greatest invention in the history of information processing** – will be with us well into the next millennium. To these numbers, we must also add a vast number of Pick-like systems: *McDonnell Douglas Reality*, *Ultimate*, *Prime Information*, *Advanced Revelation* and many others, and then we realise how strong is the Pick database model. So strong, in fact, that many of its features are coveted by those working in other environments, and there are currently at least two products – *uniVerse* and *UniData* – offering the Pick database model on Unix platforms.

There are Pick systems in almost every commercial and industrial field:

- Advertising, telesales and marketing
- Agriculture
- Building and construction industry
- Education and college administration
- Financial and accounting systems
- Health and Hospital management
- Library
- Mailing systems
- Manufacturing, costing and Bill of Materials
- Membership and registration systems
- Office administration
- Payroll and wages
- Personnel management and human resources
- Sales management and order processing
- Spreadsheets
- Stock and inventory control

You can name any branch of commerce or industry, and there is sure to be a Pick application system designed specifically for people working there.

Surprisingly, you may not have heard about the Pick system until today. So where did the Pick system come from? Where has it been hiding? Why have you not heard of it? As you will see, the Pick system is a gentle giant that has been working smoothly and quietly for a great many years

– longer than almost any other computer system – and has stayed with us whilst newer, momentarily fashionable systems with their headline-hitting births, problems and deaths have come and gone.

This is Advanced Pick introduces the Advanced Pick system and is aimed at both technical users and non-technical users. I have attempted to present the material in a manner which is suitable for those who are coming new to Pick and Advanced Pick systems, for those who are migrating from *R83 Pick* systems to Advanced Pick, and for those who have worked on Pick-like systems – such as *Reality*, *Ultimate*, *Prime Information* and *Advanced Revelation* – and who are moving to Advanced Pick.

The book starts with a brief look at the history of the Pick system. Then, for those who have already used the R83 Pick system or Pick-like systems, Chapter 2 is entitled:

Moving to Advanced Pick

and looks at individual aspects of the Pick system and summarises the main differences which will be of interest to such readers. The rest of the book takes a detailed look at the individual areas of Advanced Pick, starting with the structure of the database and going on to look at each of the tools of the system in turn.

Finally, a set of Appendices includes a glossary of terms which are likely to be encountered by anyone working in the Pick and Advanced Pick environment, a number of tables, and a chart of the <Ctrl> keys used by Advanced Pick.

A great innovation of Advanced Pick is **EPick, the Encyclopedia Pick**. This comes in two forms: a printed paperback volume and an on-line enquiry system. Both contain full explanations of all the current features of the system, all TCL commands, all Basic statements, all processing codes and much, much more. In writing **This is Advanced Pick**, it has not been my intention to replace EPick, but rather to produce a readable introduction to the major topics within the Advanced Pick system and, at the same time, drawing the readers' attention to areas which they might care to pursue further in EPick.

As I'm writing this book, Advanced Pick is in a state of flux with many features being added at each revision and each release – including SQL and GraPick – and at the same time the scope and availability of various tools are being changed. It may well be that some of the features and facilities described here are not available on your version of AP, it may be that you have others not described here. For this reason, it is important that you become familiar with *Epick* – either the book or the on-line

system – to ensure that you are using the full powers of your particular version of Advanced Pick.

You may be also interested in reading two other books which I have written for use with Pick and Advanced Pick and similar systems:

- **Access: the Pick enquiry language** (Chapman & Hall 1991) ISBN 0-412-37000-X
- **Basic: the Pick programming language** (Chapman & Hall 1994) ISBN 0-412-46660-0

both of which look at their respective topics in more detail and depth than we can go here.

Malcolm Bull

Acknowledgements

I must thank several people for their help during the conception and the gestation period of the present book:

Brian C Stone of *Pick Systems* in California, for giving me the idea and for help in drawing up a list of contents, and for his support in keeping me up-to-date with the latest AP releases.

Alan Pritchard of *ALLM Systems and Marketing* for keeping the idea going.

Glyn Yarnall, Dawn Bowley and Derek Crosson of *Pick Systems* in the United Kingdom for help and for answering my numerous questions.

Sam Anderson of *Repton Data* for help and encouragement with the production of the American edition of the book.

Then, there are a great many other people who have attended my training courses, used my **MB-Master self-tuition courses**, read my books and articles and my **MB-Guide beginner's guides** to various Pick and Advanced Pick topics, people whose questions and suggestions have indicated what interests them in their work and what areas of the system need particular attention and explanation for beginners and newcomers to the Advanced Pick system.

Conventions

I have used the terms **Advanced Pick** and **AP** interchangeably throughout the book.

The following conventions have been used for the various paradigms and examples presented in the text:

() braces have been used to enclose information which is **optional**, and which may be included or omitted, as required. The braces themselves are not typed in as a part of the statement or the command.

lower-case text

has been used to show the parameters which are to be supplied by the user. The acceptable values for these parameters will be described in the accompanying text.

UPPER-CASE TEXT

and any other characters, including parentheses and commas, are a part of the TCL commands, Basic statements and keywords and must be entered exactly as shown.

In order that the reader will recognise them as such, all commands and statements are shown in upper-case even though Advanced Pick will generally accept data and commands in upper- and/or lower-case.

Certain keyboard control keys are represented by their name enclosed in angle brackets:

<Ctrl>
<Esc>
<Return>
<Break>

Many control keys and other characters are entered at the keyboard as a combination of one or more of the above keys together with other keyboard characters. In particular, the Update Processor and other Advanced Pick tools use the <Ctrl> keys and sequences of <Ctrl> keys for cursor movement and to invoke specific actions. Throughout the book, I have shown these <Ctrl> keys in forms such as:

to represent the key which is entered by holding down the <Ctrl> key and typing the K at the same time. I have used this form in preference to the underlined notation:

k

which is used in some of the literature.

For typographical reasons, it has sometimes been necessary to break a long Access sentence or Basic statement over two or more lines:

```
SORT STOCK BY PRICE WITH LOCATION "LN" AND WITH PRICE > "50.00"
      DESCRIPTION PRICE LOCATION TOTAL VALUE
```

```
EXECUTE 'LIST STOCK WITH MATERIAL = "PINE"' CAPTURING OPUT RETURNING
      FLAGS
```

Except where otherwise indicated, each Access sentence, each Basic statement, and each TCL command should be entered on one line.

When discussing the contents of specific items on files, these will be shown as in the example here, indicating that, in this instance, the file-name is DICT STAFF, the item-id (shown on line 00) is SURNAME, and the contents of the item are shown on the subsequent lines. If no file-name or item-id are shown, these will be indicated in the accompanying text.

00	DICT STAFF
01	SURNAME
02	A
03	4
04	Employee's name
05	
06	
07	
08	
09	TX
10	10

Rather than swamp the reader in a mass of output, the data has been amended in some cases so as to reduce the output and to make more obvious the points under discussion.

To identify the various roles played by those involved with the Advanced Pick system, I have used the following terms:

- **System Manager:** for the person who will be responsible for the operational aspects of your Advanced Pick system. He/she will perform duties such as carrying out file-save and file-restores, creating new users and new accounts.
- **Systems Analyst:** for anyone who will be responsible for the functional aspects of your Advanced Pick system. He/she will perform duties such as designing and creating files, designing and creating attribute definitions.
- **Programmer:** for anyone who will be responsible for writing Basic

programs, Procs, macros, menus and other processing routines within the Advanced Pick system. In many areas, the work of the Systems Analyst and the Programmer will overlap.

- **User and End-user:** for anyone who logs on to the system to carry out his/her day to day work: inputting data, producing reports.

In a great many Advanced Pick installations, all these titles may overlap with one or two users playing all roles.

Introduction to Advanced Pick

Back in the 1960s – and even today in many computing environments – programmers spent a lot of time writing report programs and then rewriting them to meet the users' changing requirements. So common was this that software engineers addressed the situation by producing data tabulation software and report program generators.

In 1964, an American software engineer, called **Don Nelson**, joined the US organisation TRW Systems to produce an **enquiry language** which would enable users to produce reports and make enquiries about the TRW database. **Dick Pick** joined TRW in 1965 and worked alongside Nelson to develop a data model which would support the enquiry language. In March 1965, Nelson published the specification for his **Generalized Information Retrieval Language System – GIRLS**.

The processing model for the system was based upon a general-purpose computer using a general-purpose Assembly language. This language offered all the fundamental concepts of a computing system: memory access and memory handling, data storage, data manipulation and data handling, data testing, branching, addition, subtraction. This language still forms the basis of the Pick system and the set of operations still corresponds to most computers which are manufactured today. If a new processor is encountered, it is a fairly simple matter to draw up a table of correspondences between the true machine code and the general-purpose Assembly language; so simple that within two or three weeks the Pick system can be implemented on any new computer architecture which may be encountered.

By late 1965, Nelson and Pick had completed most of the enquiry language elements of the system. It was then that the TRW project expanded from a largely developmental system to a live, total systems management capability for the US Army Cheyenne helicopter project.

In April 1969, Pick left TRW with the intention of implementing the general-purpose computer model on a fairly low-cost computer, and after a brief spell with a Californian programming house, Syscom, he continued the development work alone at the computer facilities of the University of California-Irvine using a Xerox Sigma-7.

In 1972, Dick Pick set up **Pick & Associates** to apply the skills of a number of people whom he had met at UCI, Syscom, and TRW.

During 1969-1970, Pick had selected the **Microdata** (later **McDonnell Douglas Information Systems**) micro-programmable computer as a suitable platform, and in early 1972, the system was up and running. Microdata made the decision to market the system, and in 1974 the first

commercial systems were shipped under the name **Reality**.

Its ease of use and its power made the Pick system very attractive to many companies who were trying to sell business solutions for use on other hardware. In 1976, **Pick Systems** began to license other organisations to use the Pick data model as a basis for their own systems. The first of these was the **Ultimate Corporation**, selling the product for use on Honeywell Computer Systems. Other licensees followed – including **ADDs Mentor**, **Altos**, **C.Itoh**, **Fujitsu**, **General Automation**, **Prime Information** and many more – and today there are about 30 companies supplying systems which are based upon the Pick data model. More recently, the major features of the Pick system have been made available as software running within other operating systems; so you might run **Advanced Revelation** on a DOS or OS/2 machine, or **UniVerse** or **UniData** on a Unix machine. This portability means that, as an organisation's hardware requirements expand, their Pick applications can be ported to a vast range of hardware and computer systems.

The large number of companies producing versions of the system could have resulted in a range of incompatible dialects, but in the late 1970s Pick Systems produced a standardised form of the Pick system, known as **R83 Pick**, release 83. This is the basis for all the systems which are related to Pick. The major features of R83 Pick include:

- The **Access enquiry language** which allows users to produce their own reports by means of data dictionaries created for the files. As we have seen, this enquiry language was the major driving force behind the development of the Pick system.
- On-line creation and maintenance of data files.
- Multiuser support, allowing many users to work on the system at the same time.
- Simultaneous access to the same files by several users.
- Unlimited file size.
- Direct access to individual records on a file by way of the record-key.
- Flexible field and record format.
- Spooler software enabling many users to produce their printed output on one (or many) printers.
- Reading / writing data to magnetic storage media such as tape and floppy-diskette.
- The Basic language allowing users to write their own processing routines.
- The Runoff text processing system for producing text, documents and

other printed material.

The Pick system was to become one of the most powerful and attractive means of performing commercial computing, and with its Access enquiry language it represents a database management system whose power was, and still is, almost unchallenged.

1.1 Advanced Pick

Following the production and stabilisation of the R83 Pick release, Pick Systems proposed a number of radical changes to the design of the system. Indeed, so great were these changes, that they represented more than just another version of R83 Pick, and this new system – originally to be called R84 – was named **OA, Open Architecture**. During 1988, further changes were introduced to the underlying strategy of R83 and OA and the version known as **AP, Advanced Pick**, was announced. The major features of Advanced Pick are:

- **The Update Processor:** a general, full-screen editor and a dictionary-driven file maintenance utility for creating and maintaining items through the attribute definitions.
- **File indexing:** to locate an item by a *secondary key*. For example, your STAFF file may have the employee-number as the item-id, but indexing will allow you to retrieve items if you know just the employee's surname, the postcode, the telephone number, or any similar secondary key.
- **Level pushing:** to interrupt and suspend a process and then invoke a second process; when the second process is complete, you may invoke a further process at the lower level or you may return to resume the original process.
- **Macros:** to save any TCL command – or a sequence of TCL commands – and invoke these just by typing the name by which they were saved. Macros represent a simpler alternative to the facilities offered by Procs.
- **Menus:** to save a set of TCL commands and call these up – one at a time – by menu selection.
- **The TCL stacker:** to recall, to modify, and to reissue any TCL commands which you issued earlier.
- **The phantom processor:** to invoke jobs to run in the background, whilst you carry on with your normal work unhindered.
- **The Output Processor:** to produce documents and text, exploiting the range of features offered by current printer technology. This is based upon the principles of Runoff.
- **Additional processing codes** to ensure file integrity and allow the

user to pass from one file to another during file processing with the Update Processor.

- **Incremental file-save:** to perform a file-save, saving only those items which have changed since the last complete file-save was performed. The items saved by an incremental file-save can then be loaded back on to the system after a restore has been performed.
- **Transaction logging:** to log all changes made to certain files on backing storage (or store the changes until the backing storage device becomes free). The log can then be loaded back on to the system after a restore has been performed.
- An ability to interact with the host operating system, DOS and Unix.

The Pick Systems organisation still supports the R83 Pick implementation, although no further development will be carried out on that product and any work on the R83 Pick version is likely to be confined to bug-fixing. The major products from Pick Systems are now the various releases of Advanced Pick.

1.2 Implementations of Advanced Pick

Three main versions of Advanced Pick are produced by Pick systems:

- **AP/Native:** a free-standing product running on computer systems based upon Intel 286 and later micro-processors.
- **AP/DOS:** a version of AP which runs as a program on computer systems working under DOS.
- **AP/Unix:** this is actually several distinct versions, including AP/RS6000 AIX, AP/AT&T Unix, AP/SCO, AP/DGUX, which run as a process on computer systems working under the various manifestations of the Unix operating system. These implementations offer facilities which exploit the features of Unix and the C language.

These AP/Unix implementations should not be confused with systems such as **UniData** and **uniVerse** which are Pick-like emulations running on Unix platforms.

At the time of writing, Advanced Pick is available - or development is under way - on the platforms listed on Figure 1.1.

AP/DOS
AP/Native
ATT NCR 3000
Bull DPX 20
Citrix
Data General Avion
DEC Alpha Windows NT
DEC OSF/1

DG/UX
Fujitsu DS 90
Hewlett Packard 9000
HP-UX
IBM Power PC
IBM AIX 3.2
IBM RS/6000
ICL DRS 6000

Linux	Siemens Nixdorf
Motorola 88000	Silicon Graphics Indio
NCR 3000	Sinix
Novell Netware	Sony PIX 100
OS/2	Sun SPARC
Protected Mode	Sun Solaris
Sanyo Icon	Univel Unixware
SCO/Unix	Windows NT

Figure 1.1: AP platforms

1.3 Summary

For almost thirty years, the Pick system has been one of the most powerful and attractive computer systems for commercial and business applications. In the 1980s, the R83 Pick system was extended to the Open Architecture (OA) implementation, and later to the Advanced Pick (AP) system.

Advanced Pick can be installed as a stand-alone system, or as a DOS program running alongside other DOS systems, or as a Unix process running alongside other Unix systems. This spread of availability will take Advanced Pick – its database model, its enquiry language and its wide range of general commercial applications – well into the next century.

2

Moving to Advanced Pick

This Chapter is aimed at those who are migrating to Advanced Pick from R83 Pick and similar systems. We shall first look at some of the differences between R83 Pick and Advanced Pick, and then we present some of the practical points to be considered when transferring your systems to Advanced Pick.

2.1 Differences between R83 Pick and Advanced Pick

In Chapter 1, we saw that the major innovations of Advanced Pick include:

- The Update Processor
- File indexing
- Level pushing
- Macros
- Menus
- The TCL stacker
- The phantom processor
- Incremental file saves
- Transaction logging
- The Output Processor
- Additional processing codes

There are a number of changes and extensions to some of those R83 Pick features which have been carried over to Advanced Pick. Let us first look at the aspects which will be of interest to anyone migrating from R83 Pick and Pick-like systems to Advanced Pick.

2.1.1 General changes

There are a number of general points at which Advanced Pick will behave differently from your R83 Pick system. Some of these will be invisible to your users, others may not:

- Each user is individually identified by his/her own **user definition** which is held on the USERS file. Each user definition is identified by the user's **user-id**. The System Manager will be responsible for creating and maintaining these items.
- The logon activity has changed, and is now a two-step operation, requiring you to enter your own user-id (and possibly a password) and then the name of the account which is to be used (and again, possibly a password).
- A sequence of commands which is to be invoked as each user logs on may be included as a **macro** in the user definition. This can

include a LOGTO command to log you to a specific account, thus making the logon process a one-step operation as on R83 Pick. The **logon procedures** (equivalent to the *logon Procs* on R83 Pick) are still available to carry out any initialisation when a user logs to a particular account.

- The <Break> key is normally used for **level pushing**, that is, to interrupt and suspend the current process and invoke another, although it may be reassigned to call the debugger.
- The <Esc> key is normally used to send the <Esc> character, although it may be reassigned to push the processing level.
- The system debugger now uses symbolic frame addresses for the standard system routines, instead of the numeric addresses used on R83 Pick.
- The *frame size* is either 1024 or 2048 bytes, depending upon which AP implementation you are using.
- The linkage information is held in the last 24 bytes (of 1024-byte frames) or the last 48 bytes (of 2048-byte frames).
- A number of R83 Pick **user-exits** are no longer supported.
- Verbs, Access and TCL facilities. If you are migrating from a system other than R83 Pick, you should check the CLASHES file after updating your accounts for any TCL commands, Access and other verbs which are no longer available on Advanced Pick. Any such verbs should be rewritten using the AP equivalents.
- Access processing codes: R83 Pick processing codes are portable to Advanced Pick, but any non-generic codes used as Access conversions / correlatives or as Basic ICONV / OCONV operands should be checked for compatibility. This is particular important for if you are moving from systems such as *Reality* or *Ultimate* which use A;IF and AIF correlatives; these must be rewritten using the Advanced Pick IF code.

If you are using software such as *Jet*, *CompuSheet+*, *AccuPlot*, *System Builder*, *SB+* and so on, you should contact your supplier about the compatibility of your software with Advanced Pick.

2.1.2 TCL, the terminal control language

There are some differences which affect the user who is working from the TCL prompt:

- The TCL prompt is the colon.
- Any TCL command or Access sentence may be entered in either UPPER-CASE, lower-case, or mixed. Thus, the sentences:

```
sort stock
SORT STOCK
sort STOCK
SoRt StOcK
```

Sort Stock

and so on, are all equivalent. This is a potential mine-field and great care should be taken when choosing file names and item-ids and when deciding the nature of the data to be held on your files.

- A null TCL command will display the time and date before the TCL prompt. This can be enabled/disabled by the commands TIMEDATE-ON / TIMEDATE-OFF.
- There are a number of commands which are provided to enable/disable specific functions: TIMEDATE / BREAK-KEY / CASE / LEGEND / SPELLER / STACK / TCL-HDR / TYPE-AHEAD. In general, these commands may be entered with N (on) and F (off) options, or the explicit forms may be used:

TIMEDATE (N is equivalent to TIMEDATE-ON
and:

TIMEDATE (F is equivalent to TIMEDATE-OFF

respectively. The simple command:

TIMEDATE

with no options will display the current status of that function.

- A utility called R83.SETUP is provided to invoke the set of commands:

BRK-DEBUG	set the <Break> key to invoke the debugger
ESC-DATA	set <Esc> key to send the <Esc> character
SPELLER-OFF	disable the spelling checker
LEGEND-OFF	disable the legend on spooler output
TCL-HDR-OFF	disable the TCL header on spooler output
TIMEDATE-OFF	disable the time/date display at TCL
WHO (C	set the WHO output to R83 Pick format
TERM , , , , 2	set the form-feed delay to 2

and, when used in conjunction with the <Caps Lock> key, this will cause the system to behave very much like a standard R83 Pick system.

- There are duplicate definitions and abbreviated forms for many verbs and keywords both with and without hyphens. Thus, CREATE-FILE may be entered as CREATEFILE. This is made possible by having MD entries for both CREATE-FILE and CREATEFILE. However, the alternate entries are not available for all hyphenated commands, so it is probably better to stay with the full, standard and hyphenated forms.
- The format of the MD entries for the TCL verbs has changed.
- Many TCL processes are now performed by Basic programs. This means that the source code is *get-at-able* and can be inspected to see the true action of the activity, and – in extreme circumstances –

the code may even be modified for your own requirements.

- On R83 Pick, the input buffer was 140 characters in length. On Advanced Pick, the input buffer is virtually unlimited in size; this means that there is no longer any need for the **<Continuation sequence>** which had to be used when typing long Access sentences. The type-ahead buffer is also virtually unlimited.
- On AP/DOS and AP/Unix implementations, commands can be issued from Advanced Pick directly to the host operating system.
- The nature and content of the output from many commands has been changed, notably that of WHAT, WHERE, and WHO.
- A number of new TCL commands have been introduced for use with macros and menus: COMMENT / DISPLAY / PROMPT.

:ABSLOAD	CS	HALT-SYSTEM	SNAKE
ADDENDA	CURRENCY	HUSH	STRIP-SOURCE
ADDENDUM	DEL-ACC	INIT-ABS	:SWC
B/UNLOCK	DISK-USAGE	INTER	:SWD
B/DEL	DISKCOMP	LINK-WS	:SWE
B/ADD	DISCOPY	LIST-PORTS	:SWX
BEEP	DTR	OKI-192	:SWZ
CHECKFILES	EPSON	PACK	TA
CHOO-CHOO	FC	PRIQ	:TASKINIT
CLOCK	FC-ON	RTD	UNPACK
CONFIG	FC-OFF	SET-FLOP	X-REF
CREATE-BFILE	FILECOMP	SETUP.SIO	XREF
CROSS-INDEX	FKEYS	SLICE	

Figure 2.1: Some discontinued R83 Pick verbs

- A number of R83 Pick verbs and Procs are no longer supported. Some of these are shown in Figure 2.1. Note that the verbs for changing D-pointers (:SWX, :SWE and so on) are no longer available and have been superseded by the SET-DPTR command, and those used with the long-forgotten Batch Processor (B/ADD and B/DEL and B/UNLOCK verbs) have been discontinued.
- When the UPDATE-ACCOUNT process is performed to bring an R83 Pick MD up to Advanced Pick standard, any obsolete verbs and items are deleted (after copying them to the CLASHES file), and other items – such as verbs and catalogued program pointers – are updated to the AP format.
- Certain TCL commands assume the * if an item-list is omitted:

```
EDIT STOCK
COPY STOCK
CT STOCK
```

in contrast to that of R83 Pick which would require an item-list or an asterisk to be specified. However, the UPDATE command to invoke the Update Processor *does* require the * to process all items, otherwise it will create a new item with a newly-generated item-id.

- Access sentences will accept item-ids with or without quotes:

```
LIST STOCK > 1000 WITH MATERIAL "PINE"  
T-DUMP STOCK 1234 3456
```

although care must be taken if the item-ids are ambiguous and could be confused with datanames.

- The TCL stacker is standard on Advanced Pick and will store an unlimited number of TCL commands for recall and/or correction.
- All the TCL commands issued by the users may be recorded on a file for inspection. The commands – including null commands when you simply hit <Return> – are recorded on the file CAPTCL.

There are one or two points about specific TCL commands:

- The CONVERSE facility is available to communicate with another port.
- The word DATA is optional on the CLEAR-FILE command.
- The DEFINE-TERMINAL process no longer holds a definition for positioning the cursor on the current row at a given column-number.
- The DELETE-FILE command displays error-message 784 to confirm successful deletion of the file.
- The on-line HELP facility provided by the EPick (Encyclopedia Pick) can be called up by commands of the form:

```
HELP LOGTO  
or:  
? LOGTO
```

to obtain help on any TCL command, Basic statement or other Pick topic.

- Certain commands include a ? option to display a help message showing the expected format of the TCL command, although this is not yet universal.
- LOGTO / TO retain the user's spooler assignments.
- MSG command (and the alternatives MESSAGE and SEND-MESSAGE) has changed its format, and does not allow the use of value-marks to send a multiline message.
- The POVf command (and the synonyms OVERFLOW and OVf) output a summary of the space available. There are options to output the FIDs and sizes of available blocks, just the sizes of blocks, and only for FIDs in a given range.
- The TANDEM facility is available to link the current process to any other process, so that input at either port affects both.
- The printer width specified on the TERM command has been increased to 32767 characters.
- The output from the WHO command (and the U50BB user-exit) presents the port-number (p), the user-id (u) and the name of the

account (a) in the form:

p u a

and the U60BB user-exit returns:

u a

The C option on the WHO command can be used to specify that the WHO command (and the equivalent U50BB user-exit) are to present the information in the R83-compatible form:

p a u

and the U60BB user-exit in the form:

a u

Any changes to the format are associated with the user-id and are retained over LOGTO and TO commands and until the user logs off.

2.1.3 Access

The syntax of Access sentences is the same as before, and the general behaviour of the language is unchanged, although there are some new keywords and conventions:

- The use of a sort specification in a sentence automatically invokes the sorting action. Thus, a sentence such as LIST ... BY is now identical to SORT ... BY. This applies to all verbs, such as SELECT / SSELECT, LIST-ITEM / SORT-ITEM, which have a sorting form.
- Apostrophes may be omitted when specifying item-ids in an Access sentence. Thus, the forms:

```
LIST STOCK 1000 2000 DESCRIPTION
SORT STOCK > 2000 QUANTITY PRICE
T-DUMP PROGRAMS LIST.WAGES PRINT.WAGES
```

are equivalent to the more familiar sentences:

```
LIST STOCK '1000' '2000' DESCRIPTION
SORT STOCK > '2000' QUANTITY PRICE
T-DUMP PROGRAMS 'LIST.WAGES' 'PRINT.WAGES'
```

Care must be taken if the item-ids without the apostrophes are ambiguous and could be confused with datanames. If there is any conflict, a word is assumed to be a dataname. Note that it is still necessary to use quotation marks when specifying *values* in selection criteria:

```
LIST STOCK > 1000 WITH MATERIAL "PINE"
```

- On R83 Pick, the maximum length of TCL input was 140 characters. On Advanced Pick, there is virtually no limit on the length of the TCL command or Access sentence which you may type in, and for this reason ...
- The <Continuation sequence> which allowed you to type in a long sentence in short segments has been discontinued, and the special

H<< marker is no longer required when you are constructing long sentences in the Proc stack.

- The ~ character may be used in item-ids and data fields without producing unexpected results during sorts, as was the case on R83 Pick.
- The FILL modifier (and the equivalent R option) may be used with an Access sentence which is too wide for the screen (or printer) and which would otherwise produce non-columnar output. When this is used, the data is output with the headings and the data continuing across the page in a text-like manner.
- The file-name is included in the default heading for Access reports.
- There are additional options for use in HEADING and FOOTING clauses. These are available in all contexts, including Access and Basic.
- The IFNO connective has the same meaning as WITHOUT, as in:

```
LIST STOCK IFNO QUANTITY
LIST STOCK WITH NO QUANTITY
LIST STOCK WITHOUT QUANTITY
```

- The relational operators EQUAL and IS are offered as synonyms for = and EQ:

```
LIST STOCK WITH MATERIAL = "PINE"
LIST STOCK IF MATERIAL IS "PINE"
LIST STOCK WITH MATERIAL EQUAL "PINE"
```

The word ARE is still a throwaway modifier.

- The LEGEND facility allows the System Manager to specify a piece of text which is to appear at the foot of every Access report *which is sent to the printer*. The text of this system-wide legend is specified (in Update Processor format) in an item with the item-id LEGEND which is held on the MESSAGES file. If required, legends may be created for individual users.

The inclusion of a legend on Access reports is enabled/disabled by the LEGEND-ON / LEGEND-OFF commands at TCL for all subsequent Access sentences, and by the LEG-SUPP modifier (or the equivalent K option) for an individual sentence.

- The *ITEMS LISTED* message has changed and shows how many items were listed and how many items are on the file:

```
123 items listed out of 300 items.
```

- The NI-SUPP modifier (or the equivalent B option) suppresses the *ITEMS LISTED* message at the end of an Access report. The equivalent S option (which was available on R83 Pick 3.1) has been withdrawn.
- The REFORMAT and SREFORMAT verbs use the original item-ids for the new items (unless ID-SUPP is specified, in which case the first

output field is used as the new item-id, as on R83 Pick). A further difference is that if the item already exists on the output file, the new data is added as multivalues (R83 Pick overwrites any existing items).

- The ROLL-ON modifier (and the alternative TOTAL-ON) behaves very much like the BREAK-ON modifier, except that it does not use a separate column to show the data; instead it *rolls* the output data into the following column or into the previous column wherever possible.
- The SAMPLING modifier restricts an Access report to a specific number of items. For example, the sentence:

LIST STOCK SAMPLING 20

will display only the first twenty items on the file.

- The SS – spreadsheet – connective produces a tabular summary of the values of one attribute between specific dates held in another attribute.
- When an Access sentence produces a report *on the printer*, the sentence is normally shown at the head of the report. This action may be enabled/disabled by the TCL-HDR-ON / TCL-HDR-OFF commands at TCL for all subsequent Access sentences, and by the TCL-SUPP modifier for an individual sentence.
- The letter X may be used in conjunction with the L / R / T attribute type codes in an attribute definition, and will cause that column to be expanded to fill the report width in an Access report.
- Attributes 11 and above of the system definitions (user definitions, file definitions, data-level identifiers, attribute definitions) are used for a variety of purposes, many of these are associated with the Update Processor.

Advanced Pick offers additional facilities for handling the **output list** and the **implicit output list** (or the **default output list**) in an Access sentence. On R83 Pick systems, a sentence such as:

LIST STOCK

with no output list, will automatically include numeric datanames 1, 2, 3 and so on, in the sentence, if these are available; this set of numeric datanames comprise what is known as the implicit output list or the default output list. This feature is still supported on Advanced Pick but there are further extensions:

- Attribute 15 of the data-level identifier may hold a list of datanames which are to be used in the absence of any other output specification. This macro specification is also used by the Update Processor. Any list held here is used in preference to the datanames 1, 2, 3, 4 and so on.
- The ONLY modifier suppresses the use of the macro and displays only the item-ids.
- **Predefined phrases** can be defined on the DICT of a file and used

in Access sentences. The special phrases @ and &filename can be used as an alternative to the implicit output list facilities.

For compatibility, the attribute definitions *A0, *A1 and so on, are still available, although they have been superseded by a new feature known as **temporary attribute items** which allows you to use names such as A0, A1, A2, A3 and so on (indefinitely), in Access sentences. These are similar to the *A1, *A2 and *A3 names, except that the items A1, A2 and A3 do not exist, they are constructed for temporary use by the Access processor.

2.1.4 Select-lists / saved-lists

There are several changes to the ways in which **select-lists** and **saved-lists** are handled.

Saved-lists may now be held on any file. Thus, the SAVE-LIST command has the form:

```
SAVE-LIST {{file.name} list.name}      '
```

If the file-name is omitted, for example:

```
SAVE-LIST XXXXX
```

then the POINTER-FILE is assumed, as on R83 Pick. If the file-name and the list-name are omitted:

```
SAVE-LIST
```

then the name:

```
%u
```

is used, where *u* is the user-id.

The other commands for handling saved-lists may also include the file-name parameter. Note that, since DC-files are no longer required, POINTER-FILE is a normal file.

An important feature is the concept of the **primary select-list** and the **secondary select-list**. If a select-list is created in the familiar manner:

```
SELECT ...
SSELECT ...
GET-LIST ...
QSELECT ...
```

then this is considered to be the **primary** list. If the S option is included in the command (or the SECONDARY keyword in an Access sentence), then the list is considered to be the **secondary** list. Certain TCL verbs are able to use these two lists as the input list and the output list respectively. For example, the sequence:

```

SSELECT FILEA = "A]"
SSELECT FILEB = "Z]" (S
COPY FILEA (OD
TO: (FILEB

```

will select all items on FILEA starting with A and use these (the primary list) as the item-list on the COPY command, and will then select all items on FILEB starting with Z and use these (the secondary list) as the item-list on the (FILEB destination line. Other commands which can use the primary and secondary lists are:

```

COMPARE
COMPARE-LIST

```

The Basic READNEXT statement has been extended:

```

READNEXT key SECONDARY ELSE STOP

```

thus allowing a program to pick up keys from the secondary list and handle two select-lists at the same time.

There are several new verbs for handling select-lists and saved-lists:

- o COMPARE-LIST to compare the contents of two lists
- o SORT-LIST to sort the keys in a saved-list.
- o FL to combine the keys in two saved-lists.
- o ISELECT / SSELECT to select all the items which are in the same physical group as a specific item.
- o NSELECT to produce a list of those items in the current select-list which are *not* in a given file.
- o The J option on SELECT / SSELECT sentences will suppress the display of the messages.

There are several synonyms for the familiar verbs for handling select-lists and saved-lists:

- o CL for COPY-LIST.
- o DL for DELETE-LIST.
- o EDIT-LIST invokes the standard Pick line-Editor to edit a saved-list, whereas:
- o EL invokes the Update Processor to edit a saved-list.
- o GL for GET-LIST.
- o LL for LIST-LISTS.
- o SL for SAVE-LIST.

Some of these abbreviated forms were available on R83 Pick and other implementations.

2.1.5 Processing codes

The processing codes of R83 Pick are available as before, but Advanced Pick introduces a number of additional codes. Many of these are used in the familiar manner, as *conversion codes* and *correlatives* in attribute definitions, and in the Basic *ICONV* and *OCNV* functions, whilst others are now encountered in file definitions and data-level identifiers, but several are provided especially for use with the *Update Processor* to control the nature and format of changes made to the data on the files. The list of new processing codes is shown in Figure 2.2.

B	FT	MC/AN	MCS	O	XC	XR
CALL	I	MC/NA	MI	V	XD	XS
CU	ID	MCAN	MP	X	XI	XT
DF	IF	MCNA	MY	XA	XO	XTx

Figure 2.2: New processing codes

2.1.6 Basic programming

There are many extensions and changes to the Basic language. Let's look first at the physical program:

- It is no longer necessary to have the DICT section of a program file defined with a DC-pointer. A Basic program can be held and compiled on any file which has DICT and a data section. For this reason, the CREATE-BFILE verb has been discontinued.
- Any Basic programs transferred from R83 Pick to AP must be recompiled before they can be used on AP.
- The format of catalogue-pointers on the MD has changed.
- Blank lines may appear within the source program. Labels without a statement are also permitted:

```
DESTINATION:
10
```

- There is no conflict between statement labels and variables names. Some releases of R83 Pick would compile successfully but then report a fatal run-time error if a statement label and a variable name were the same.
- The \$INCLUDE compiler directive is offered as an alternative to INCLUDE.
- The INSERT / \$INSERT compiler directives are provided.
- A program may use up to 214,748,362 variables and/or array elements at one time. If you have insufficient disk space, you may find that too many variables will abort the program at run-time.
- A string may be up to 2 gigabytes (2,147,483,648 characters) in length.
- Variable names may be up to 48 characters in length, and, on later

releases, the underscore character may be used.

- o There are two verbs to compile a Basic program: the BASIC command is identical to the R83 Pick compiler where all keywords and variable names used in the source code are *case-sensitive*, so that the names *TAX* and *Tax* and *tax* would identify three different variables; the COMPILE command is *case-insensitive*, and the names *TAX* and *Tax* and *tax* would identify the same variable. Apart from the case-sensitivity of the source code, the two compilers are identical.
- o On AP/Unix systems, a new version of Basic is available. Known as **Flash/Basic**, this produces programs in the object code for the specific machine and is therefore significantly faster than standard Basic code which is interpreted at run-time.
- o The B/LIST command is used to indent a Basic source program, replacing BLIST of R83 Pick.
- o A number of additional verbs are available: COMPILE to compile a case-insensitive Basic program; COMPILE-CATALOG to compile and then catalog a Basic program or subroutine; COMPILE-RUN to compile and then execute a Basic program.
- o The RUN-LIST command will execute, one after another, a sequence of programs submitted as a select-list. This might be used in a sequence such as:

```
SSELECT WAGES.PROGRAMS = "YEAR.END]"
RUN-LIST WAGES.PROGRAMS
```

There are several new Basic statements and functions although some of these – such as the IN and OUT statements – may be familiar to programmers who have worked with Release 3.1 PC Pick.

- o The operators ^ and ** are available for exponentiation.
- o The \ operator is used to return the remainder after integer division.
- o A feature, known as **operator assignment**, is available, allowing you to use forms such as:

```
A += 123
```

as an alternative to the familiar:

```
A = A + 123
```

- o The following functions are new: ACCESS; ASSIGNED; ERROR; FOLD; OCCURS; SORT; SOUNDEX; SUM.
- o The following statements are new: \$INCLUDE; \$INSERT; CAPTURING; CASING; CLOSE; labelled COMMON; CONVERT; DEL; ERROR; EXIT; FILE; GET; IN; INS; KEY; LET; MATBUILD; MATPARSE; OUT; READTL; READTX; REPLACE; ROOT; SEND; TCL.
- o The range of @ function calls has been extended. Many of the new

features can be used to produce special effects on the printer.

- On those systems which work in DOS or Unix environments, there is a range of statements – of the form %xxx – which will allow a Basic program to communicate with those environments and handle their files. These include %OPEN, %READ, %WRITE and %CLOSE statements.
- The MATREADU, READU, READVU, RELEASE, and WRITEU statements use item locks instead of group locks.
- **Substring assignment** can be used to insert a substring within a string. An extended *field store* version of this feature resembles the Access Group extraction code and replaces one or more groups of data within a string.
- The ACCESS function is provided to access data within subroutines which are called by Access attribute definitions.
- The ASSIGNED function can be used to determine whether or not a variable has been assigned a value.
- The CASING ON/OFF statements can be used to specify whether data strings, including keyboard input data, is to be case-sensitive or not. If CASING is OFF, then the three data values YES, Yes and yes are considered to be identical.
- The CLOSE statement has been provided to close a file which is no longer to be accessible to the program.
- Labelled COMMON blocks of data, such as:

```
COMMON /STAFF/ NAME, AGE, RATE, CLOCK.NO
```

can be used in subroutines which handle just a subset of the COMMON data used by a program and its subroutines.

- The CONVERT statement translates the contents of a string according to two control strings.
- The DEL and INS statements are offered for handling dynamic arrays. These were already available on some non-Pick systems. The statement:

```
DEL REC<1,2,3>
```

is equivalent to the familiar:

```
REC = DELETE(REC,1,2,3)
```

and the statement:

```
INS 'ABC' BEFORE REC<1,2,3>
```

is equivalent to the familiar:

```
REC = INSERT(REC,1,2,3; 'ABC')
```

- Dimensioned arrays must be declared *before* they are used. R83 Pick would allow a DIMENSION statement to be placed anywhere within the program.

- o An array may be dimensioned at run-time:

```

SIZE = 10
DIMENSION DIMARR(SIZE)
OR:
INPUT PEOPLE
DIMENSION NAMES(PEOPLE)
INPUT YEARS
DIMENSION STATS(PEOPLE,YEARS)

```

- o Dimensioned arrays defined within the COMMON area must be declared with an explicit numeric value for the size.
- o When passing a dimensioned array as an argument to an external subroutine, it is not necessary to use the MAT identifier, and within the subroutine an array can be declared with no dimension:

```

SUBROUTINE SUB001(PASSARRAY, PASSB, PASSC)
DIMENSION PASSARRAY()

```

provided that it is declared explicitly in the calling program.

- o When handling dimensioned arrays and dynamic arrays in logical expressions, subscript references may use a *wild card* character in statements such as:

```

IF LIST(*) = 'Y' THEN ...
IF TABLE(4,*) > 1000 THEN ...
IF STRING(*,*,*) < 0 THEN ...
IF STRING<1,2,*> = '' THEN ...

```

to take the specified action if *any* element of the array satisfies the condition.

- o The ERROR statement can be used to display a message from the MESSAGES (or ERRMSG) file, rather like the STOP statement, except that the ERROR statement does not terminate the program.
- o The ERROR() function returns the TCL command which invoked the program, rather like the new TCLREAD statement.
- o The EXECUTE statement will accept a sequence of TCL commands separated by attribute-marks:

```

EXECUTE 'RUN BP REPORT' : CHAR(254) : 'OFF'

```

- o The EXIT statement leaves the current loop by jumping to the statement immediately following the next physical NEXT or REPEAT statement.
- o The FILE statement is used to open a file, declare a suitable dimensioned array and accept data references of the form STOCK(QUANTITY) which use Access datanames in the coding of a Basic program.
- o The FOLD statement breaks a string down into substrings of a specific length.
- o The GET statement gets a string of ASCII characters from a port

which has been attached by means of the DEV-ATT command. The GETX statement returns a hexadecimal string. The SEND statement will send a string (or the binary equivalent if SENDX is used) to a port which has been attached by means of the DEV-ATT command.

- The HEADING/FOOTING statements have an extended range of options.
- The IN and OUT statements are provided for single character I/O, as on Release 3.1 R83 Pick.
- The IN and INPUT statements have extended forms:

```
IN variable {FOR time {THEN / ELSE statements}}
```

```
INPUT variable { ,length} {:} {_}  
  {FOR time {THEN / ELSE statements}}
```

with an optional time-out specification for the time for which the processor is to wait for the user's input. If data is entered within this time limit, the THEN clause is taken; if no data is entered with the time limit, the ELSE clause is taken.

- The KEY statement is used to search through a B-tree index to find a key which matches (or is nearest to) a given search string.
- The LET keyword is optional on assignment statements:

```
LET A = 100
```

- An indefinite loop of the form:

```
LOOP  
  statement(s)  
REPEAT
```

may be constructed. The EXIT statement is available to leave the loop.

- The MATBUILD statement uses the contents of a dimensioned array to construct a dynamic array:

```
MATBUILD dynarr FROM dimarr USING CHAR(254)
```

- The MATPARSE statement uses the contents of a dynamic array to load the elements of a dimensioned array:

```
MATPARSE dimarr FROM dynarr  
  {USING CHAR(254)} SETTING count.variable
```

- The regular assignment statement can also be used to transfer data – element by element – between a dimensioned array and a dynamic array. For example:

```
dimarr = dynarr  
dynarr = dimarr
```

- The OCCURS function searches a dynamic array for a value which occurs consecutively for at least a specific number of times.

- The statements which handle the backing storage device may specify the ONERR clause as an alternative to the ELSE clause. This clause is taken if there is an error during the execution of the statement and returns a code via SYSTEM(0) indicating the nature of the error.
- The THEN/ELSE clauses may be omitted on the OPEN statement. A form such as:

```
OPEN 'STOCK'
```

is equivalent to the familiar form:

```
OPEN 'STOCK' ELSE STOP 201, 'STOCK'
```

- The PRECISION statement will allow a precision in the range 0 to 9. The default is still 4.
- The THEN/ELSE clauses may be omitted on the various READ statements. A form such as:

```
READ S.REC FROM ITEM.ID
```

is equivalent to the familiar form:

```
READ S.REC FROM ITEM.ID ELSE S.REC = ''
```

- The READTL statement will read the label on the backing storage device.
- The READTX command will read a record from backing storage and pass this to the program as a hexadecimal string. This means that records which include Pick and Basic control characters – such as CHAR(255) – can be processed. The WRITETX is not available.
- The READNEXT statement has a new keyword:

```
READNEXT key {SECONDARY} ...
```

to pick up successive items from a **secondary** select-list generated by a SELECT sentence which included the S (for Secondary) option. This allows a program to handle two select-lists at the same time.

- The REPLACE statement is used within a *bridge* link to change an item-id.
- The ROOT statement finds the root of a B-tree index, and the KEY statement subsequently uses this root to find within the index a specific item-id which matches a given argument. The KEY statement has options to return the item-id which matches the argument, the next sequential item-id, or the previous item-id,
- The Basic SELECT statement may specify a dynamic array instead of a file-variable. This will cause the READNEXT statement to pick up successive attributes from the dynamic array:

```
LIST = 'A':CHAR(254):'B':CHAR(254):'C'
SELECT LIST TO LIST.VAR
LOOP
  READNEXT LETTER FROM LIST.VAR ELSE EXIT
REPEAT
```

- The SORT function sorts the attributes (or values) of a dynamic array into ascending, left-justified sequence.
- The SOUNDEx function returns a four-digit code equivalent to a specified string.
- The SUM function will add together all the adjacent subvalues of a dynamic array and replace these with the result as a single value; if there are no subvalues, then it will add together all the adjacent values to produce a single attribute; if there are no adjacent values, then it will add together all the attributes to return a single numeric result.
- The SYSTEM function has been extended.
- The TCL statement executes a specific TCL command and, optionally, returns the error-message numbers and parameters of the messages generated by the command as the elements of a dimensioned array. The format of the statement is:

```
TCL tcl.command {to dimensioned.array}
```

The general effect of the TCL statement is similar to that of the EXECUTE statement, except that there is no direct facility equivalent to the CAPTURING clause of the EXECUTE statement. As with the EXECUTE statement, a sequence of several TCL commands may be invoked if the individual commands are separated by attribute-marks.

- The TCLREAD statement will pass the entire TCL command which invoked the program into a specified variable.

Lists of the full range of Basic statements and Basic functions are shown in the Appendix.

2.1.7 Accounts, files and items

There are a number of changes to the ways in which accounts, files and items are handled on Advanced Pick:

- The names of certain system accounts have been changed; for example, DM (instead of SYSPROG). In all cases, synonyms are supplied to make the older names accessible.
- Early AP releases offered several new accounts: PA, a general personal administration account; QA, for system testing and reporting problems to Pick Systems; TUTOR, a training account demonstrating certain new features. These are not essential to the use of Advanced Pick and can be deleted from your system.
- The PROCLIB account has been discontinued.
- The names of certain system files have been changed; for example, MDS (instead of SYSTEM); ERRORS (SYSTEM-ERRORS); MESSAGES (ERRMSG); FILE-OF-FILES (STAT-FILE). In all cases, synonyms are supplied to make the older names accessible.
- The text of the messages in the MESSAGES file has been modified

slightly.

- The nature of the POINTER-FILE has been demystified. It is now a regular file and the items and saved-lists which are held there can be handled just like items on any other file. The list-handling commands – COPY-LIST, EDIT-LIST, and DELETE-LIST – are still supported.
- File indexes are supported, allowing items to be retrieved via any of a number of indexes based upon the item-id and/or any combination of the data within the item.
- The dictionary code in attribute 1 of D-pointers (account definitions, file definitions and data-level identifiers) has been extended: L indicates that all changes are to be logged by the transaction logger; S indicates that item-ids are case-sensitive. C has been discontinued (since files with DC-pointers are redundant).
- D-pointers cannot be handled by the EDITor. The Update Processor can be used to modify D-pointers, and a new SET-DPTR verb is available to add/remove letters from attribute 1 of a D-pointer.
- The FILE-OF-FILES file plays all the roles of the STAT-FILE on R83 Pick. It also records every file created and/or restored to the system, and when and by whom a file is created, cleared, and restored.
- Advanced Pick uses a different procedure for handling locks on files / groups / items.
- The values for the modulo and file size are unlimited.
- The concept of **separation** has been discontinued. All files have an implied separation of 1.
- The item size is unlimited.
- Each physical item is preceded by an 8-byte control field. This indicates the length of the item, the nature of the item (whether it is a D-pointer or not), whether it is held directly or indirectly, and whether it has been created, changed and/or deleted since the last file-save.
- As items are created / changed / deleted, a flag (called the **dirty bits**) is set within the 8-byte control field to indicate this. When an incremental file-save is performed, only these flagged items will be saved. When a full file-save is performed, the dirty bits are cleared.
- Items less than 847 bytes in length are stored in the file space (as on R83 Pick), but longer items are held indirectly in frames of virtual memory with just a pointer to those frames being held in the file space, much like items on DC files on R83 Pick.
- The structure of attribute definitions has been extended and more attributes are available. Many of these are used by the Update Processor to control the nature and format of the data which is entered into a field. The general form of attribute definitions for use with Access is unchanged.
- File synonyms, Q-pointers, are set by the familiar SET-FILE command. This has a third element – the local name by which the Q-

pointer is to be established on the MD.

- **Pathnames** – similar to the root/directory structure of DOS and Unix – can be used to access files on other accounts.
- It is now possible to have a Q-pointer which points to a Q-pointer. This allows the System Manager to move files around much more easily and more closely approaches the **data transparency** requirement of a database.

2.1.8 Backing storage

There are a small number of differences in the way in which backing storage devices are handled on Advanced Pick:

- There are several new commands for handling magnetic tape and/or floppy diskette.
- The SET-DEVICE command is used to select the appropriate backing storage device on AP/Unix systems.
- The SET-FLOPPY options have changed.
- On release 6.1 and later, the **multiuser interface** allows a user to attach two or more devices, and allows two or more users to handle the various devices simultaneously. To support this, there have been changes to some of the commands described below: T-DET is used to detach one or all devices from the current user, and T-ACT is provided to switch between two or more devices which are attached to the current user.

2.1.9 File-save and file-restore

There are several changes to the ways in which you can perform your save and restore operations:

- Later versions of AP use the :FILES command to invoke a full file-restore. Earlier versions perform a file-restore by coldstarting the system with the systems diskettes loaded and then taking the appropriate option.
- The R83 Pick and Advanced Pick T-DUMP / T-LOAD files are compatible in both directions, and R83 Pick file-save / account-save files can be used for account-restore and/or selective-restore purposes on Advanced Pick, though Advanced Pick file-save / account-save files cannot be used to pass data to R83 Pick implementations.
- The file-save will process a select-list of accounts to be saved. Thus, a sequence such as:

```
SELECT MDS 'WAGES' 'PAYROLL' 'PAY-STATS'  
FILE-SAVE
```

will save just the accounts WAGES, PAYROLL, PAY-STATS (and the DM account). This is equivalent to the **multiple account-save** of

some Pick-like systems.

- o Advanced Pick offers an **incremental file-save** facility which, during a file-save operation, will dump only those items which have changed since the last complete file-save. The FILE-SAVE utility (and the equivalent forms of the SAVE verb) now ask:

Is this an incremental save?

and then proceed in the familiar manner. The ACCOUNT-SAVE utility is unchanged.

- o Each physical item is preceded by an 8-byte control field which records whether the item has been created, changed and/or deleted since the last file-save. When an incremental file-save is performed, only these flagged items will be saved. When a full file-save is performed, the *dirty bits* are cleared.
- o The STAT-FILE has been superseded by the FILE-OF-FILES file.
- o The **transaction logging** feature records all changes made to certain files. This transaction log is written directly to backing storage (or held until a device is available), and may be restored to the system after a file-restore / account-restore has been performed.
- o There is a facility on the SAVE verb to save only those items whose item-ids are held in attribute 25 onwards of the item on the FILE-OF-FILES file which relates to the file being processed. If attribute 25 contains an asterisk, then all items will be saved.
- o A new ABSDUMP verb is available to save just the ABS file on backing storage.
- o The SAVE verb has several new options, many of these are associated with the incremental save facility.
- o The display produced during the save and the restore processes has been changed, showing:

account.name > file.name > data.name

although there is an option on the SAVE verb to display the names in the R83 Pick hierarchical format:

- o The account-name which is specified on the ACCOUNT-RESTORE command will be assumed if a null response is given to the NAME OF ACCOUNT ON DISK message during an account-restore.
- o The RESTORE-ACCOUNTS utility will read through a file-save diskette/tape, restoring any accounts which are not on the system.

2.1.10 Spooler

The general features and use of the spooler are essentially unchanged. The following points are of interest:

- o There is no limit to the number of jobs which can be handled by the spooler.

- Jobs are identified by the user-id instead of the name of the account which submitted the job.
- The L option has been discontinued with the STARTSPOOLER command.
- The ASSIGNFQ command:

ASSIGNFQ q p

associates form-queue number *q* with printer *p*. There will be a definition for each printer on the DEVICES file.

- A number of additional options are available with the SP-EDIT command and with the SP-KILL command.

2.1.11 Runoff and the Output Processor

Runoff is still supported on Advanced Pick, but it has largely been superseded by the Output Processor. The following points will be of interest if you are likely to move Runoff documents from R83 Pick systems to Advanced Pick, or if you are considering using the Output Processor:

- Advanced Pick Runoff is compatible with that of R83 Pick. Some differences are:
 - The default page width is taken from the terminal characteristics and is not 60 as on R83 Pick.
 - The layout of HEADINGS and FOOTINGS may need adjusting.
- Although Runoff and the Output Processor are similar, the two are not compatible and Runoff documents cannot be handled by the Output Processor because, in general, Runoff commands are not compatible with those of the Output Processor.
- Output Processor commands may be embedded anywhere within the text. It is not necessary to hold them on separate command lines, as is the case with Runoff commands.
- All active commands, such as .JUSTIFY, are disabled by a corresponding X-form .XJUSTIFY.
- Output Processor macros allow you to create a sequence of commands and other information which can be called just like a regular Output Processor command.
- There are facilities for automatic numbering of figures and tables.
- There is a greater recognition of the features of current printers, with facilities for handling subscripts, superscripts and fonts.

2.2 Converting from R83 to AP

A great many organisations will be moving from their existing R83 Pick system to a new Advanced Pick system; others may be moving to Advanced Pick from Pick-like systems – ADDS, C.Itoh, Fujitsu, McDonnell

Douglas, Ultimate and so on. The actions to be taken are essentially the same in all cases:

Step 1: Make magnetic copies of all the data and programs on your existing system in a form which can be read by your Advanced Pick system.

For R83 Pick systems, you can make a file-save, a set of account-saves or T-DUMP copies of the files. For others, you may be able to use file-save, account-saves. If in doubt, ask your supplier. In all cases, T-DUMP files are compatible and can be used to transfer any data; this may be tedious but it is fairly easy to automate the procedure by writing a simple Basic program to produce the T-DUMPs and to read them back on to AP later.

Make sure that your magnetic copies are reliable. To be sure, make two copies of everything.

If your old system is to be available alongside the Advanced Pick system, it may be possible to use proprietary software, such as *Mainlink*, *PK Harmony*, *Termite*, or *Viaduct*, or even the Advanced Pick IMPORT command, to transfer the data.

Since your Basic programs will have to be recompiled when they have been moved to Advanced Pick, there is no need to retain the object programs of your old system.

Step 2: Install Advanced Pick.

Because of the variation in the mechanical aspects of installing Advanced Pick, we shall not discuss this here. You will be supplied with full details of the steps to be taken when installing your particular implementation of AP.

Step 3: Load the data files from your old system on your Advanced Pick system.

This may be done in several ways:

- Restore all accounts by performing RESTORE-ACCOUNTS from a file-save tape. Use the form:

RESTORE-ACCOUNTS (n

where *n* is the size of frames on the old system. The size option will instruct AP to make the appropriate adjustments to the modulus of the files as they are restored.

- Restore individual accounts by performing an ACCOUNT-RESTORE from a file-save tape or an account-save tape. Use the form:

ACCOUNT-RESTORE aaaa (n

where *aaaa* is the name of the new account, and *n* is the size of frames on the old system.

You may also need the C option if you are restoring from a Compatible system (such as McDonnell Douglas or Ultimate).

- Restore individual files by performing SEL-RESTORE from a file-save tape or an account-save tape, and/or ...
- Restore individual files by performing T-LOAD from a T-DUMP tape.

Remember that, in the case of SEL-RESTORE and T-LOAD, you must first create the files which are to receive the imported data, and, when you do this, the frame size on your old system may not be the same as on Advanced Pick, and you will need a smaller modulo (if your old frame size was smaller than AP) or a larger modulo (if your old frame size was larger than AP).

Note especially that:

- An R83 Pick file-save tape cannot be used to perform a file-restore on the AP system.
- You should not attempt to restore the system accounts or files such as:

SYSTEM
ACC
BLOCK-CONVERT
EXECUTE-CONTROL
PROCLIB
SYSPROG

SYSTEM-ERRORS

and so on, as they already have their counterparts on Advanced Pick. If these are inadvertently restored, you should delete them later.

Step 4: For each account restored to the system, perform the UPDATE-ACCOUNT process.

This uses the contents of the Advanced Pick NEWAC file to convert the verbs and other items on the MD of an account to the AP form. The process is invoked from the DM account by a command of the form:

```
UPDATE-ACCOUNT aaaa
```

or:

```
UPDATE-MD aaaa
```

to update the MD of account *aaaa*. If the account name *aaaa* is omitted, the process will ask:

```
Enter md file name:
```

If a select-list is available, as with a sequence such as:

```
GET-LIST xxxx
UPDATE-ACCOUNT
```

then a number of accounts may be updated at one time. The process first asks:

```
Create user id from each account name (y, <n>)?
```

which will allow you to create a user-id equivalent to the account being updated. The new user definition includes a standard setup sequence of TCL commands:

```
SP-ASSIGN
BRK-DEBUG
ESC-DATA
SET-SYM GSYM
```

followed by a LOGTO the new account.

The UPDATE-ACCOUNT process scans the MD of the account and imposes the new format on the standard items, and deletes certain obsolete items (CHOO-CHOO goes!) and invalid verb items; the original items are retained on a newly-created data section (aaaa) of the CLASHES file. Procs are retained and catalogued program pointers are converted to the AP format.

Step 5: For each account, produce a list of the CLASHES,aaaa file and check whether any of the changes will affect your applications.

This is particularly important if you are migrating from a system other than R83 Pick, on which you may have been using features and facilities which are not available on generic Pick.

Step 6: Check the compatibility of the imported items.

In the first part of this Chapter, we looked at some of the differences between R83 Pick and AP systems, and this information should help you here, particularly in those areas where R83 Pick facilities have been changed or discontinued. In general, you should have no problems importing true data files, but particular attention should be paid to the processing items:

- **Basic programs:** These must be recompiled before they can be used on Advanced Pick.

R83 Pick Basic source code is completely compatible with AP Basic, and in such cases, you will simply have to recompile your Basic programs. However, if you are migrating from a system other than R83 Pick, where your programs may have used features and facilities which are not available on Advanced Pick, you will have to check the source code and rewrite any non-standard coding in Advanced Pick Basic. Particularly important are any uses of:

@(-n) functions
EXECUTE / PERFORM statements
INCLUDE statements
SYSTEM functions

and any functions such as GROUPSTORE, FIELDSTORE which are not available on AP. Lists of the AP Basic functions and Basic statements can be found in the Appendix.

When you recompile your programs, you should remember that the BASIC verb is case-sensitive (recognising the names *TOTALS*, *Totals* and *totals* as three distinct variables), whereas the COMPILE verb is case-insensitive (regarding all three names as the same variable). If your programs use lower-case variable names for any special purposes, you should use the BASIC verb.

- **R83 Pick Procs** are completely compatible with Advanced Pick.
Procs for systems such as McDonnell Douglas and RPL will have to be rewritten if they use features and facilities which are not available on generic Pick. A list of AP Proc statements can be found in the Appendix.
- **Assembly language routines.** These will have to be rewritten in almost all cases. On Advanced Pick, the Assembler and Assembly language is now offered as a separate product.
- **User-exits** are also included under this heading and you should check

that those used in your systems are supported on Advanced Pick. A number of user-exits from R83 Pick – particularly those related to low-level interfacing with the hardware and the disk – are no longer supported.

- If you are using software such as *Jet*, *CompuSheet+*, *Accu/Plot*, *System Builder*, *SB+* and so on, you will have to contact your supplier about the compatibility with Advanced Pick.

Step 7: Check any incompatible or unsupported facilities in your old system, such as verbs, Access and TCL facilities, and rewrite these in Advanced Pick terms, if necessary.

If you are migrating from a system other than R83 Pick, you should check the CLASHES file for any TCL commands, Access features, and verbs which are not available on Advanced Pick. Any such verbs should be rewritten using the AP equivalents.

- Access processing codes

R83 Pick processing codes are identical to those of AP. If you are moving from other than R83 Pick, any processing codes used as Access conversions / correlatives or as Basic ICONV / OCONV operands should be checked for compatibility. This is particular important for those systems which use A;IF and AIF correlatives; these must be rewritten using the Advanced Pick IF code.

Make any changes which your old system may expect of the standard files and routines:

- SYSTEM: there may be account synonyms and Q-pointers, the logon message, and other items which you wish to carry over from your old system.
- The coldstart routines are held as macros SYSTEM-COLDSTART and COLDSTART on the MD of the DM account.
- PROCLIB: this is not used on Advanced Pick. You may need to create and restore this account/file to support any Procs used by your system.
- BP: this file has new versions of many programs. You may need to inspect your old SYSPROG BP file to transfer any discontinued programs which your old system still requires.
- CURSOR: this file has been replaced by the DEVICES file. Check that all your terminal types and printers are supported by the DEVICES file. Use the DEFINE-TERMINAL command to maintain / define any additional devices which you may need.

Step 8: Make any changes to support your users.

This will include such facilities as:

- Set the date to the correct format for your country.
- Set up the appropriate terminal characteristics. Use the SET-TERM command to establish the default characteristics; create/amend the DEVICE field of the PIBS items to include the terminal characteristics for each port:

UPDATE PIBS 'p' DEVICE

where *p* is the port-number. Ensure that the MACRO field of each user definition on the USERS file includes a TERM-TYPE command to invoke these terminal characteristics.

- Create a user definition identified by a unique user-id for each user. There may be conflict (with the TCL stacker, for example) if two or more users log on with the same user-id.
- Set the MACRO field of the user definitions on the USERS file to include any appropriate initialisation as each user logs on. You may wish to include a LOGTO command in the MACRO field so that each user is automatically logged to a specific account and the logon procedure resembles that of the R83 Pick system.

Your new Advanced Pick system should now be ready for use.

2.3 Summary

In general, R83 Pick and similar Pick-like systems can be transferred to your Advanced Pick system without too much effort. When an R83 Pick account has been restored to AP, the UPDATE-ACCOUNT command will make all the necessary changes to the MD of the account. There may be instances, however, in which the effects of certain R83 Pick processes have been changed or withdrawn completely.

3

System architecture

Advanced Pick is available on a wide range of hardware. Some systems are based on personal computers and serve only one user, others support several users, whilst others run on mini-computers and will handle many hundreds of terminals.

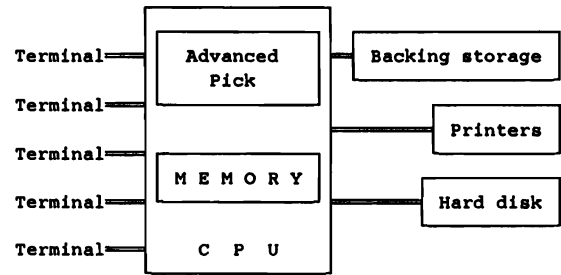


Figure 3.1: AP/Native

Figure 3.1 shows a schematic view of a typical system running under AP/Native. Your entire computer and its hard disk may be dedicated to Advanced Pick, or you may be running AP/Native in one partition whilst R83 Pick or DOS occupy other partitions of the same disk.

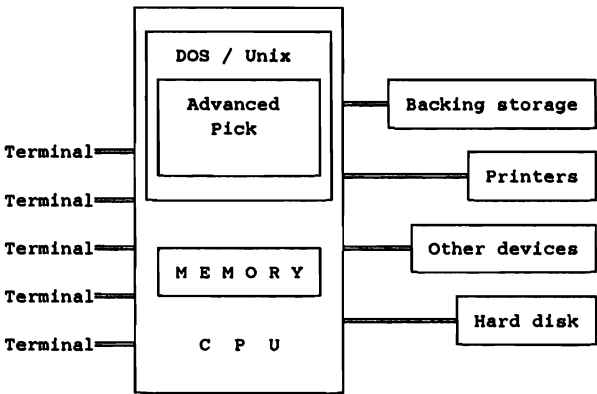


Figure 3.2: AP/DOS and AP/Unix

If you are using a version of Advanced Pick which runs under the DOS or Unix operating systems, then the physical picture is slightly different, as

illustrated in Figure 3.2. Here, Advanced Pick runs just like any other DOS or Unix program, alongside other DOS and Unix programs, and – through the native operating system – has full access to the range of hardware and software which is available on that system.

Whatever its nature and whatever its size, every Advanced Pick system is made up of the same basic components:

- 1) **The CPU:** the central processor unit and the system itself.
- 2) **The memory or RAM:** this holds all the data and programs which are being processed by the various users at any moment.
- 3) **The user terminals:** Advanced Pick is designed to be used by several users at one time, each user working independently of the others at his/her own terminal. Each terminal is connected to the computer via a *port*. The actual number of ports, and the devices which are attached, depends upon the particular configuration of the system which you are using. Advanced Pick will support a wide range of terminal equipment, including VDUs, printer-terminals and terminals with slave printers, but many features of the system are designed specifically for VDU terminals and screen displays.
- 4) **The printers:** any number of printers – serial and parallel – may be connected to the system. A part of the system software, known as the *spooler*, handles all printed output and enables two or more users to generate printed reports at the same time.
- 5) **The backing storage device:** this is used to pass data from one system to another and to produce back-up copies of the system and its files for security purposes. The backing storage devices may be any combination of magnetic tape(s) and/or floppy diskette(s).
- 6) **The hard disk / fixed disk:** all the programs and the data files which are created and processed by the individual users and those used by Advanced Pick itself are held permanently on a hard disk. This disk storage space is shared by all users of the system, and the actual nature and size of the available disk space depends upon the configuration of your particular system. The *virtual memory* feature of Advanced Pick means that this disk space is regarded as an extension of the memory space.

Based upon this fundamental configuration, it is perfectly feasible to attach other equipment and devices such as bar-code readers, plotters, point-of-sales terminals, fax and telex machines.

Let's look at the components of your Advanced Pick system.

3.1 Ports

A **port** is the connection between a user's terminal and Advanced Pick. Physically, it's simply a socket at the back of the machine.

When you buy an Advanced Pick system, you specify the number of ports which you need. A 1-port system (sometimes called a 1-user system) can only have one user at a time, a 6-port system may have up to six simultaneous users, and so on.

Each port is identified by a port-number: 0, 1, 2 and so on. There are many TCL commands which use the port-number, rather than the user-id or the name of the account which is using that port.

Although not exactly equivalent, the terms **line** and **PIB** are often used with the same meaning as *port*.

Advanced Pick will work with virtually any sort of terminal device, and you will find dumb terminals – such as Wyse 30 – and intelligent terminals – such as PCs, and although most equipment now comprises screen and keyboard, you may even find old printing terminals. It is also possible to link specialised equipment, such as point-of-sales cash registers, to your Advanced Pick system.

With the increasing presence of PCs in offices and workshops, many organisations have begun to use these as terminals to their Pick systems. This is more cost-effective than buying dumb terminals for the same purpose. The PC operates with a *terminal emulation* program running under DOS. This DOS program then sends signals to, and receives signals from, the Pick system. As far as the Pick system is concerned, the PC is behaving exactly like a terminal, but the emulation software allows full access to the features of the PC and of DOS systems in general. As you will see in the computer press, there are many terminal emulation packages available, including *PK Harmony*, *Termite* and *Viaduct*.

3.2 Printers

Two types of **printer interface** are used: *serial* and *parallel*. Any printer can be attached to any suitable port on the Pick system, and any number of printers may be linked to the system, provided that there are ports available. The STARTPTR command is used to declare a particular port as a printer and to associate a specific output queue with that printer.

Slave printers can be attached to any terminal which is provided with an auxiliary output port. When the slave printer is activated, all output sent to the screen may also be sent to the slave printer. The reference manuals will give details of the control sequences used to control your slave printer.

3.3 Hard disk

The most important part of the hardware on the Advanced Pick system is the **hard disk**. The **virtual memory** feature of the system views the hard disk as an extension of memory (or RAM), and allows the users to access any part of the database as and when required.

In physical terms, the hard disk comprises one or more circular metal plates mounted on a central axis and sealed inside an airtight unit. Each surface of these plates is viewed as a series of concentric **tracks**. Each track is divided into a number of **sectors**. When you buy your hardware, you will specify how much disk space you require. When you are installing the system, you can specify exactly how much of this disk space is to be made available to Advanced Pick.

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
		28529	28530	28531
28532	28533	28534	28535	28536

Figure 3.3: The frames of virtual memory

3.3.1 Frames

In logical terms, Advanced Pick views the disk space as a sequence of sectors or **frames**. Each frame is identified by a unique number called the **FID**, the **frame identifier**. Thus, the first sector on the first track of surface 0 of the disk is considered to be frame 0 (that is, FID 0), the next sector on that track is frame 1, and so on. Thus, if we had 28537 frames dedicated to our system, then we could visualise the hard disk as being made up of an array of frames, as illustrated in Figure 3.3. The highest-numbered frame would then be frame 28536 or FID 28536.

This is an idealised situation. In practice, it may be that only a part of the hard disk is dedicated to Advanced Pick, as on AP/DOS which shares the

disk with DOS, or there may be faulty tracks on your disk and Advanced Pick would skip these when drawing up its map of the frames of the disk.

The actual size of the frames depends upon which implementation of Advanced Pick you are using. Some use 1024-byte frames (1000 bytes for data and 24 bytes of control information), others use 2048-byte frames (2000 bytes for data and 48 bytes of control information). The WHAT command displays the size of the frames on your system, as you can see in Figure 3.5 (a).

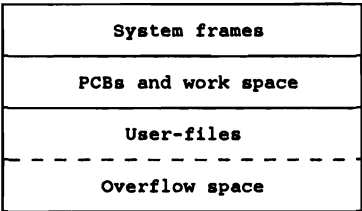


Figure 3.4: Virtual memory

The disk space is organised into four areas for use by the system itself and to hold all the data and programs which the users will handle, as shown in Figure 3.4.

- **System frames:** this is the area in which the programs for all the standard Advanced Pick software are held; these are usually known as the **ABS frames** (*absolute*, because they are always held in the same position). Special PICKWARE products, such as *Accu/Plot*, *CompuSheet+*, and *Jet*, may also use these frames for their own special routines, and a number of frames are set aside for users to load their own assembly language processing routines. This system area typically occupies about 2048 frames.
- **PCBs and work space:** the primary control blocks and the work space for each available port hold registers and other control information relating to each individual port on the system. Each user has control of the system for a short period of time (a **time-slice**) during which he/she can do whatever work they wish. At the end of the time-slice, that user's work is suspended and the next user is given a time-slice. The PCBs and the work space store information about what each port is doing and enable each user to continue his/her work when their next time-slice comes around. In Figure 3.5 (a), we see that the PCB for port 0 starts at FID 2048.
- **User-files:** next comes that area of disk which holds the data files used by the system and the users. The MDS file holds details about all accounts on the system and is normally the first file in this area, and is followed by the MD for the individual accounts and their files. The start of this area is known by the symbolic name **SYSBASE** and this is where the MDS file begins. In Figure 3.5 (a), we see that the MDS file starts at FID 2394.

- o **Overflow space:** beyond the area occupied by the files comes the (as yet) free and unused area of disk which will be used whenever new files are to be created and when an existing file requires more frames to accommodate additional items and as files and items grow in size. Look at the report shown in Figure 3.5 (a); here, we see that, when this report was produced, there were 8234 frames available for overflow out of a total 28537 frames.

There is no fixed boundary between the user-files area and the overflow space. When a file-restore operation has been performed, the disk is fairly tidy with all the user-files grouped together and one large chunk of overflow space following. However, in time, the user-files take over parts of the overflow space as new files are created and as existing files expand, and overflow frames occur in the user-file space as existing files and items are deleted.

There are several commands for monitoring disk usage and the fragmentation of the disk space, and for interrogating other details about your system. One of these, the WHAT command, will produce a report such as that illustrated in Figure 3.5 (a). This tells us that on this particular system:

- o The size of the memory (RAM) is 1664K,
- o There are 74 PIBs.
- o The PCB for port 0 starts at frame 2048.
- o The SYSBASE (the MDS file) is at frame 2394 and has a modulo of 7.
- o There are 28536 frames on the hard disk, of which ...
- o 8234 frames are currently unused and available for creating new files and for the growth and overflow of existing files.
- o The data frames are 1000 bytes (or 1024 bytes, to be precise) in size.

memory	pibs	pcb0	sysbase/mod	maxfid	available	dfsize
1664k	74	2048	2394	7	28536	8234 1000

(a) WHAT report

6398- 6398 :	1	6401- 6405 :	5	6407- 6412 :	6
6414- 6414 :	1	6431- 6437 :	7	6439- 6439 :	1
6441- 6444 :	4	6446- 7482 :	1037	7663- 7664 :	2
8996- 8998 :	3	9555- 9560 :	6	9564- 9566 :	3
9584- 9607 :	24	10610-10610 :	1	10618-10618 :	1
10620-10620 :	1	10634-10634 :	1	13857-13857 :	1

14664-14664 :	1	14714-14714 :	1	14716-14716 :	1
14718-14718 :	1	14720-14720 :	1	14724-14724 :	1
14728-14728 :	1	14745-14745 :	1	14753-14753 :	1
14769-14769 :	1	14772-14772 :	1	14775-14775 :	1
14833-14838 :	6	14844-14844 :	1	14874-15301 :	428
15303-15303 :	1	15305-15306 :	2	21858-28536 :	6679
overflow: 8234 reserve: 1024 blocks: 36					

(b) OVERFLOW report

Figure 3.5: Looking at your system

The OVERFLOW command displays information about the disk space, giving a more detailed picture than the WHAT command and indicating the fragmentation of the available space. A typical report is shown in Figure 3.5 (b). This tells us that there are 8234 frames of hard disk space available for the creation of new files and for the overflow of existing files and that these are in 36 separate blocks. If we inspect these blocks we see that frame 6398 is not used and is available for use in creating a new file or if an existing file grows, the five frames 6401 to 6495 are not used, and so on.

When looking for disk space to create a new file, the system will scan the table of available frames, looking for the smallest gap into which the file can be placed, and then allocate these frames to the file and remove them from the table. As an existing file grows and needs more disk space, this will be grabbed, frame by frame, from the pool of available frames.

If the available disk space becomes too fragmented and there are a great number of small blocks available, it may prevent the creation of new files and may make overflow of existing files inefficient. In such circumstances, a file-save and file-restore should be carried out.

The reserve space of 1024 frames shown on the OVERFLOW report is set aside for emergencies and can be controlled by the SET-OVF-RESERVE command.

3.3.2 Virtual memory

All data and programs are held on the hard disk. This includes the standard files and the software provided with your Advanced Pick system, and also the data and programs which the users create for their own use. Whenever you wish to execute a program, change a data record or simply inspect a data record – each program and each piece of data must

be brought into memory (RAM) before it can be used. Advanced Pick uses a **virtual memory** technique to make the information on hard disk available in memory when needed by the users.

Just as the hard disk is organised as a sequence of frames, the memory is organised as a sequence of buffers. Data is transferred between disk and memory a frame at a time; each frame is loaded into a memory buffer, and Advanced Pick keeps a record of which frame is in which buffer.

Let us imagine that you wish to carry out the wages calculations for your company. You, the user, simply type in a command indicating the name of the file of programs and the name of the particular program which you wish to use:

```
RUN PROGRAMS WAGES
```

Advanced Pick then consults its files to discover the disk location of the PROGRAMS file, then it does a calculation to determine the frame at which the WAGES program is to be found; finally, the required frame(s) are read from disk and copied into memory, and the WAGES program can then be called into action. The same thing happens if you give a command such as:

```
CT STAFF 1234
```

to inspect the record for employee number 1234 on the STAFF file (we'll see later that the CT command copies a record to the terminal screen): Advanced Pick consults its files to discover the disk location of the STAFF file, then it does a calculation to determine the frame at which the record for employee 1234 is to be found; finally, the required frames are read from disk and copied into memory, and from there the required record can be picked out and displayed on the screen.

As you and the other users carry out their work, frames of information are continually being read into memory and held there for inspection and amendment by the users.

In order to improve the speed of the system, frames are not written back to disk each time they are changed or when they are no longer required; instead, only updated frames are written to disk, and this is done only when memory becomes filled with changed frames and space must be made for new frames of data to be brought into memory from the hard disk. All updated frames are also written to disk when the system is closed down.

In some situations, there may be a large amount of data in memory which has been updated in memory but not yet written back to disk. For this reason, the machine must be properly shutdown and not just switched off, otherwise the frames in memory would show any changes we had made,

but the original frames on disk would not. To minimise the risk of losing data in the event of a power failure, when there is a quiet period with no user activity, Advanced Pick utilises the time by copying any changed frames back to disk.

A simple analogy of the virtual memory system is that of a monk [*a user or a program*] studying [*processing*] in a library. On the shelves [*the hard disk*] is a collection of pages [*frames*]. The monk needs to refer to various pages of information during his study, and as he does so he goes to the library shelf, and brings a page of information to his desk [*memory or RAM*]. As he works, he occasionally makes corrections [*updates*] to the pages on his disk. Later, the monk needs a further page of information, he goes to the shelf and brings this to his desk. Unfortunately, his desk is not very big. He can only hold a certain number of pages there at one time. He eventually reaches the situation where his desk surface is filled with pages and he needs yet another page from the library. He fetches the new page from the shelf, and looks around the pages on his desk to see which of the pages already on the desk top has been used least often, having found one, he returns this old page the shelf and puts the new page in its place on the top of the desk. If he finds that he needs a page which is already lying on his desk, he does not, of course, fetch it from the shelf a second time.

To represent more closely the true Pick situation, our analogy must be modified slightly: instead of bringing the *actual* page from the shelf to the desk, the monk makes a *copy* of the page (leaving the original on the shelf) and brings the copy to his desk [*he reads the frame from disk into memory*]. Now, he need only return an old page to the shelf [*write a frame back from memory to disk*] if he has made any corrections to it whilst it was on his desk top.

In practice, there will be several monks studying at the same time [*several users processing simultaneously*]. They all share the same desk top and all have access to the pages which they and the other monks have fetched from the shelves. Throughout their study, the pages on the desk [*the frames in memory*] are continually being swapped between the shelves and the desk.

Each time a new frame of information is required by a user and has to be brought from disk to memory, Advanced Pick must perform a fairly complex series of operations:

- 1) The system must check to see whether the required frame is already in memory.
- 2) If the frame is currently held in memory, then no further action need be taken, and two or more users can read from (and even write to) the same frame at the same time.

- 3) If the frame is not in memory, the system must fetch the frame from disk. This is a time-consuming activity.
- 4) The system must then choose a memory buffer where the new frame can be loaded into memory. The first choice will be to use a memory buffer which has not been changed.
- 5) If there are no unchanged memory buffers, the system will choose a changed buffer and write this back to the appropriate frame of disk before going on to use that buffer. Again, this is a time-consuming activity.
- 6) Finally, the system will copy the new frame into the chosen memory buffer.

This act of fetching a new frame from disk is known as a **frame fault**. As you see, a frame fault involves a complex sequence of events and can take an appreciable time to perform. The Systems Analyst who is designing files and programs will attempt to minimise the number of frame faults which are required and thereby speed up the processing.

You can see this process for yourself if you issue a command such as:

COUNT STOCK

to look at a file which you have not used today. The light on the disk drive will flash furiously and you will have to wait a moment as the frames which hold the STOCK file are read into memory. If you then issue the same command again, there will be very little disk activity and you will get the answer almost immediately, since the frames which hold the data of the STOCK file are already in memory.

All those who are using the Advanced Pick system at any one time are placed in a queue, first come, first served; the user at the head of the queue is allowed full control of the system when his/her turn comes around. Typically, each user gets a **time-slice** or **quantum** of a few milliseconds in which to carry out some of his/her work. This is the maximum time which is allowed for each process before it is interrupted and placed at the end of the queue of users waiting for their turn. A time-slice may be interrupted and the user placed at the end of the queue in other circumstances:

- A **frame fault** may occur in which the user's process has requested data which has to be read from disk. The process can only continue when the required data is in memory and the time-slice is abandoned – allowing some other user to have a go – until that time.
- The process may have attempted to read a frame which is currently being updated and must wait until the write is complete.

- The process may have issued a SLEEP command to give up its time-slice.
- The process may have executed a command which expects the user to enter data at the keyboard and can only continue when the user has entered that data.

Despite this complicated process of swapping frames in and out of memory, and of moving users to and from the queue, each individual user has the impression that he/she has sole and sustained control of the computer. It's just that the work seems slower at some times, and faster at others.

3.4 Backing storage

In addition to the hard disk, your Advanced Pick system may be fitted with one or more **magnetic storage devices**, such as:

- 3.5" floppy diskette,
- 5.25" floppy diskette,
- 0.5" magnetic tape,
- 8mm magnetic tape, and/or
- 0.25" streamer cartridge tape.

We use the term **backing storage** to refer to these devices because they are primarily used to make *backup* copies of the contents of the hard disk; you'll make these copies by means of the file-save and the account-save utilities. Magnetic tape and diskette can also be used to archive data – such as last year's invoices – which are no longer required on the live system, and to transfer data between one Pick system and another. There are a number of TCL commands and other facilities for handling the backing storage devices, as we shall see in Chapter 20.

3.5 Summary

The hardware associated with a typical Advanced Pick system includes the hard disk, one or more terminals, one or more printers, and one or more backing storage devices.

The hard disk is used to hold all the data and files which are needed by Advanced Pick and the users. The sectors on the hard disk are interpreted as frames on the Pick virtual memory system, and are brought into memory as and when required.

The backing storage devices are used to make backup copies of the system, to hold data which is not required on the live system, and to transfer data to other Pick systems.



File structure

All the information which you and your users will process on Advanced Pick is organised as a set of **files**. You will use several files in your work: there may be a STAFF file to hold details of the company's employees; a CLIENT file to hold details of the company's customers, a SALES-LEDGER file to hold details of the orders which your clients have placed for your company's products and services, a BOUGHT-LEDGER file, a SUPPLIER file, and many more. There may be any number of files on your system.

Each file holds a number of individual **items** or *records*. Each item can be created, looked at, changed, and/or deleted quite independently of the other items on the file. Thus, the STAFF file may have one item for employee number 1234 (John Dark), one for employee 2009 (Naren Josh), another for employee 2531 (Mary Wilkinson), and so on. There may be any number of items on a file. Figure 4.1 shows how we might visualise a file as a set of items.

Within each file, each item is uniquely identified by its **item-id**. So, if I want to look at John Dark's record, I would ask for the item which is identified by item-id 1234. The item-id is a unique identifier for that item. Typically, the items on the STAFF file would use the employee number as their item-id, and the items on the CLIENT file might use the client account number as their item-id. The Systems Analyst who designs the file will decide what information is to be used as the item-id of each file.

Each item is made up of a number of **attributes** or *fields*. Thus, the STAFF file items in Figure 4.1 contain one attribute for the surname, one attribute for the given names, one for the data of birth, and so on. The Systems Analyst who designs the file will decide what attributes will be held, and how these attributes are to be organised within each item. In Figure 4.1 and Figure 4.2, the first attribute of the STAFF file is the

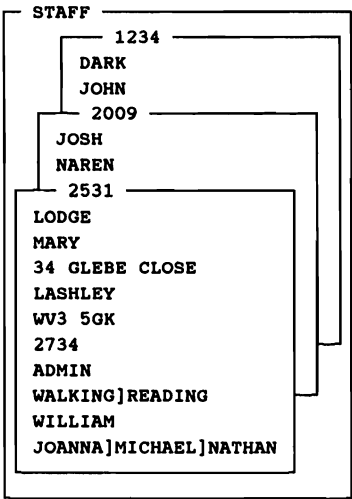


Figure 4.1: A file of staff information

employee's surname, the second holds the employee's given names, and so on. There may be any number of attributes in an item. All the items on any particular file will have the same number of attributes and these will always be in the same sequence.

4.1 The physical organisation

Now, let's see how this information is recorded physically on the hard disk.

4.1.1 The physical items

Figure 4.2 shows two items on the STAFF file. Figure 4.2 (b) shows how these would be held, each as a string of characters on the disk.

00 Item-id:	1234	2531
01 Surname:	DARK	LODGE
02 Given:	JOHN	MARY
03 Street:	10 THE OAKS	34 GLEBE CLOSE
04 Town:	EPTON	LASHLEY
05 Postcode:	WV10 1AB	WV3 5GK
06 Telephone:		2734
07 Department:	LAB	ADMIN
08 Hobbies:	CARS	WALKING]READING
09 Spouse:	ANN	WILLIAM
10 Children:	JOB	MARK]MICHAEL]NATHAN

(a) The logical view of the items

1234^DARK^JOHN^10 THE OAKS^EPTON^WV10 1AB^^LAB^CARS^ANN^JOB^_	
2531^LODGE^MARY^34 GLEBE CLOSE^LASHLEY^WV3 5GK^2734^ADMIN^	WALKING]READING^WILLIAM^MARK]MICHAEL]NATHAN^_

(b) The physical appearance of the items

Figure 4.2: Items on a file

As you see, each attribute may be as long or as short as necessary. The individual attribute which make up each item are separated by the **attribute-mark**, shown here as the ^ character.

If a certain employee did not have, say, a telephone number, then this particular attribute – as attribute 6 of item 1234 here – would be empty or **null**. Note that there is still an attribute-mark for this null attribute.

Some attributes, such as the hobbies (attribute 8) and the names of the

children (attribute 10), may have more than one **value**. Such an attribute is said to be a **multivalued attribute**. The individual values which make up the attribute are separated by the **value-mark**, shown here as the] character.

It is further possible for each value to be made up of a number of **subvalues**. With the value, the individual subvalues are separated by the **subvalue-mark**. However, these subvalues are rarely used.

The **attribute-mark** is the ASCII character 254; the **value-mark**, character 253, and the **subvalue-mark**, character 252. A special marker – the **end-of-item marker** – consisting of the attribute-mark followed by the **segment-mark** (character 255) indicates the end of each physical item. As a double-check on the consistency of the data, each item is preceded by a **control field** which records the length of the item, as we shall see later.

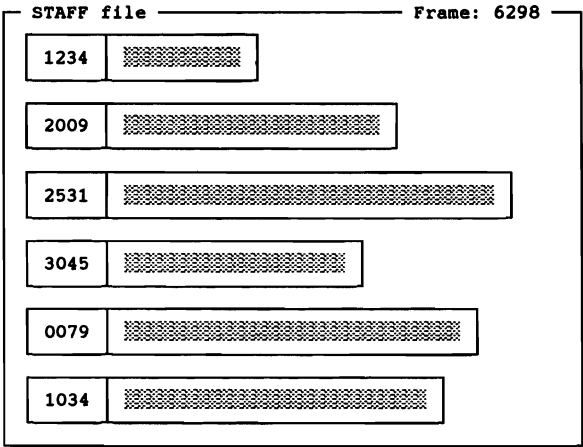


Figure 4.3: Items on a file – the physical view

When the System Analyst creates the STAFF file, he/she will indicate how much space is required for the file on the disk; a file may occupy one or more frames of disk space. Advanced Pick will then search for a block of disk space of the required size, and set these frames aside for sole use by the STAFF file. Let's imagine that the analyst has asked for just one frame of disk space, and that Advanced Pick has looked through the list of available frames – like that in Figure 3.5 (b) – and allocated the frame 6298 for the STAFF file. The system will then create an item on the MD – the master dictionary – which indicates that the STAFF file is to be found at frame 6298. The STAFF items may then be created and added to the STAFF file; this will normally be done by an application program or by the Update Processor. We can think of the items as separate strings of data held in the frame of disk space which has been allocated to the file, as in Figure 4.3; the shaded area represents the data in that item.

A technical note: If we were to represent the control field mentioned

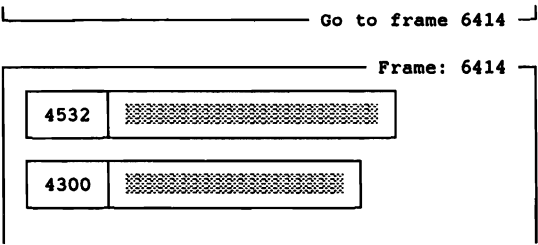


Figure 4.5: The file overflows

Here we see the original frame 6298 and then just the first part of the overflow frame 6414 which contains the new items 4532 and 4300; the rest of the frame is unused. You see that the number of the following overflow frame is recorded at the end of the original frame.

As we add more and more items to our file, this overflowing can continue indefinitely, as illustrated in Figure 4.6 (c) of the next section. This means that, if we have a large number of items on the file, it will require a large number of frames to hold the data, and each one of these will have to be brought into memory and scanned when we are looking for an item and when we are changing an item; as we saw when we were talking about frame faults, this could take a long time.

Let's see how Advanced Pick will allow us to overcome this.

4.1.2 The physical file

I have mentioned that, as with our STAFF file, the smallest possible file consists of a single frame, and we can picture this as in Figure 4.6 (a). This is known as the **base frame** or the **primary file space**.

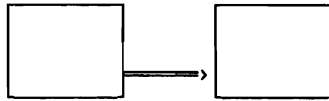
Although the physical frame is actually 1024 bytes in size (or 2048 bytes on some systems), only 1000 (or 2000) bytes of each frame are used to record the user's data. The final 24 (or 48) bytes of each frame are used by Advanced Pick, as we shall see in a moment.

As items are added to the file and the items themselves increase in size, the data will take up more and more of the available space, and there may not be sufficient space in the frame to accommodate the data. When this happens, Advanced Pick looks around for an unused frame and writes the remainder of the data in this **overflow** frame, setting **links** or **pointers** which enable the system to navigate from the base frame to the overflow frame. This is illustrated in Figure 4.6 (b).

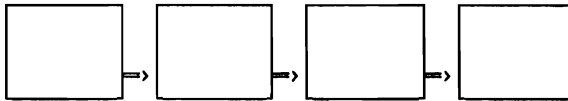
As the file grows further, more and more overflow frames will be added to the end, each pointing to the next, as you can see in Figure 4.6 (c).



(a) A single frame file



(b) The file overflows



(c) The file overflows further

Figure 4.6: A file overflows

Each frame in a sequence such as Figure 4.6 (b) and (c) contains a link to the following frame (known as the **forward link**), if there is one, and a link back to the previous frame (known as the **backward link**), if there is one. The forward links, which tell Advanced Pick where to go next, and the backward links, which tell Advanced Pick where it has come from, are recorded in the final 24 (or 48) bytes of each frame. I haven't shown the backward links in these diagrams, but we shall see them later when we look at the actual frames which hold the items.

The process of attaching more and more frames can be continued indefinitely and enables the file to accommodate as many items as are needed, but – because Advanced Pick does not keep any sort of index to show which items are held in which particular frame – each time you need to find an item, you have to start at the base frame and read all the way through all the intervening frames until you find the item you need. So on average, you have to scan half the items on the file to find any specific item, and if you are creating a new item, then you have to read all the way through the file, just to be sure that the item doesn't already exist, and then to append the new item to the end of the file.

Each time Advanced Pick has to move through a file, it has to read the first frame, pick up the linkage information, move to the next overflow frame and so on until the frame holding the required item is found. Since it is only possible to read one frame at a time, this is very time-consuming. The more items there are, the longer it takes.

To minimise this problem, Advanced Pick extends this model for a file. Instead of having just a single frame or a single group of frames as we have in Figure 4.6, a file may consist of several **groups** of frames. A file may be created with any number of groups. The file in Figure 4.7 consists of three groups. The number of groups in the file is known as the **modulo** of the file; so Figure 4.7 shows a file with a modulo of 3. The groups are conventionally numbered from 0: 0, 1, and 2, in this instance.

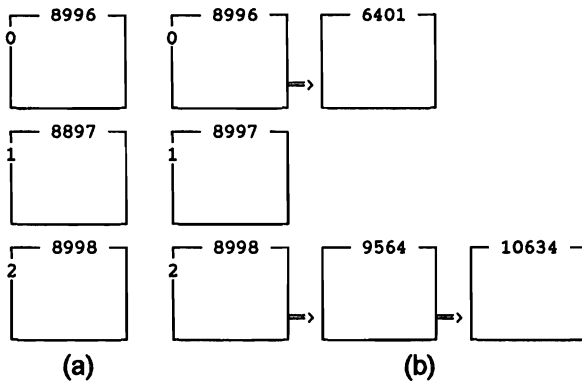


Figure 4.7: A file with a modulo of 3

When the Systems Analyst creates a file, he/she will specify the name of the file and the modulo of the file; Advanced Pick will then look for a block of, in this case, 3 contiguous frames and set these aside for the sole use of the file. Let's imagine that the system found that the three frames 8996, 8997 and 8998 were available, then the picture might look like that in Figure 4.7 (a). There would also be created an item on the MD indicating the name of the file, the number of the first frame in the file (8996), and the modulo of the file (3); this item enables Advanced Pick to find the physical location of the file.

How does Advanced Pick decide where to hold each item? When an item has to be written to the file, Advanced Pick takes the item-id and performs an arithmetic calculation on it. The process is called **hashing**, and the result of the calculation will be 0 or 1 or 2. According to this result, that particular item will be written into group 0 or group 1 or group 2 of the file; each item is allocated into one of the three groups. When an item has to be *read* from the file, the same hashing calculation is performed and the contents of the appropriate group (0 or 1 or 2) are scanned for the required item, as in the earlier single-group model.

All Advanced Pick files are organised in this way, the single-group model which we discussed first is simply a special case of a file with a modulo of 1.

As more items are added to the file in Figure 4.7, and as existing items grow, each group will overflow as necessary to accommodate the items which **hash** into that group. Each group behaves rather like a separate, independent physical file, and more overflow frames are tacked on to each group as the space occupied by the items in the group increases. The overflow behaviour of one group is quite independent of that of the other groups in the same file. Indeed, in any particular file, some groups may have many overflow frames whilst other groups – in the same file – may have no overflow at all and some groups may even be empty! In Figure 4.7 (b), we see that the first group of the file overflows into one more frame (FID 6401), the second group doesn't overflow at all, and the third group overflows first into frame 9564, and this overflows further into

frame 10634. Do you see how the overflow frames are scattered about the disk? It's just a matter of whereabouts on disk Advanced Pick can find a single, unused frame when it needs an overflow frame.

For the reasons mentioned above, the ideal situation is that there should be no overflow in the file and that Advanced Pick only has to read one frame in order to find a particular item; this means that the items can be accessed as quickly as possible. One solution to this is to create the file with a suitably large modulo. In general, the more items there are in the file, the larger will be the modulo. We discuss the process of choosing modulus for files in Chapter 5.

4.1.3 The hashing algorithm

In order to determine the group in which a particular item should be placed, Advanced Pick performs a calculation known as the **hashing algorithm**.

To illustrate the algorithm, let us imagine that we wish to create an item on a file with a modulo of 3. Advanced Pick looks at the item-id of the item and the individual characters of the item-id are converted to their numeric equivalents, based on the standard ASCII table, then weighted according to their position within the item-id, and then their sum is divided by the modulo for the file. The remainder indicates the group where that item is to go; a remainder of 0 refers to group 0, the first group in the file, a remainder of 1 refers to group 1, the second group in the file, and so on.

Let's illustrate the hashing algorithm by considering the way in which it handles item-id 1234, as illustrated in Figure 4.8. The character 1 is converted to its ASCII numeric value (49) and multiplied by 1000 (because there are three characters remaining in the item-id) and this gives a result of 49000, then the next character 2 is converted to its ASCII value (50) and multiplied by 100 (giving 5000); then 3 is converted to 51 and multiplied by 10 (giving 510); then 4 is converted to 52 and multiplied by 1 (giving 52). The intermediate results are added together, giving 54562. Dividing by 3 gives a *remainder* of 1, directing us to group 1 of the file.

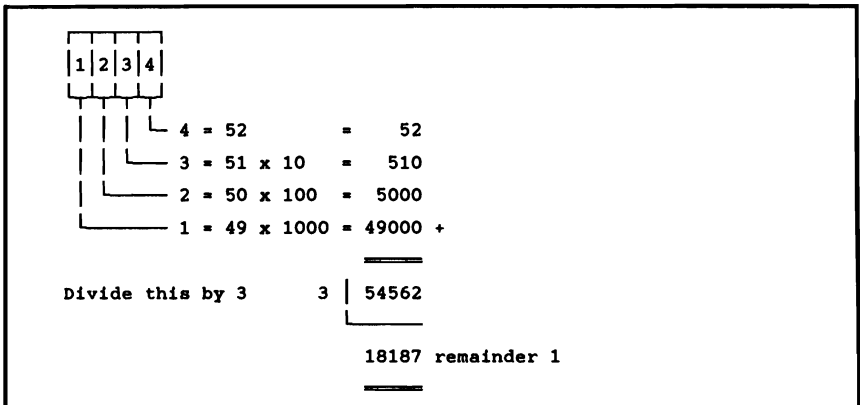


Figure 4.8: The hashing algorithm

The same process is applied if we have a longer item-id or if the item-id includes characters other than letters and numbers, although letters are converted to their **lower-case** forms. For example, if we were hashing the item-id **AB/9000** we should use character a (97) times 1000000 + character b (98) times 100000 + character / (47) times 10000 + character 9 (57) times 1000 + character 0 (48) times 100 + character 0 (48) times 10 + character 0 (48) gives a total of 107332328. Dividing by 3 gives a remainder of 2, allocating item AB/9000 to group 2 of the file.

In practice, this means that, when I issue a command such as:

```
UPDATE STAFF 1234
```

Advanced Pick will first look to the MD for the item relating to the STAFF file. Let's imagine that this shows that the STAFF file begins at frame 8996 and that it has a modulo of 3, as in Figure 4.7. Having found the modulo, the system can carry out the hashing algorithm, as above, and find that the item 1234 should be in group 1. Adding this 1 to the base frame FID (8996), Advanced Pick knows that it must then read frame 8997 and scan through the group of frames starting there to find item 1234.

On Advanced Pick, the algorithm always works in the manner described here; it is unchanging and unchangeable. This means that you cannot modify, say, the weighting factor of 10, nor can you make any other changes to the algorithm, either for a particular file or for all your files.

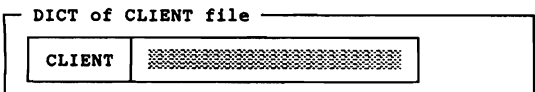
Incidentally, although my examples use item-ids 1234 and AB/9000, it is most likely that all the item-ids on a file will be of the same form. Thus, the STAFF file may have item-ids such as 1234, 0079, 4300; a BOOKINGS file may have item-ids such as 95/1, 95/77, 95/9101 and so on.

4.2 The structure of files

On all Pick systems, a file is more than just a section of disk which holds the data. There are other entities and concepts which are associated with the file space itself.

4.2.1 Dictionaries

In most cases, a file will consist of two parts: the **dictionary section** and the **data section**. To the user, the more important of these is the data section which holds the true data of the file. The other part of the file is the dictionary section – usually known as the **DICT**. The DICT section of a file normally holds the items which define the vocabulary and the datanames which will be used in Access sentences which handle the data section of the file.



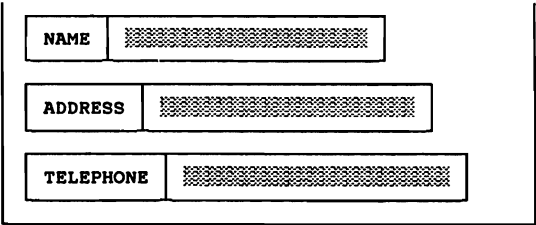


Figure 4.9: The items on the DICT section of a file

For example, if we wanted to produce a report about the data on the CLIENT file, we might issue an Access sentence such as:

```
LIST CLIENT NAME ADDRESS TELEPHONE
```

In this case, we should require definitions for the datanames **NAME**, **ADDRESS** and **TELEPHONE**, and these would be held on the DICT section of the CLIENT file. The DICT section of the CLIENT file might look like Figure 4.9. These dataname definitions are items very much like those in Figure 4.3; they consist of attributes and values. The item with the item-id CLIENT is used by Advanced Pick itself. We shall look at the format of these items later.

Both the DICT section and the data section are set up when the Systems Analyst creates the file. This might be done with a command such as:

```
CREATE-FILE CLIENT 1 3
```

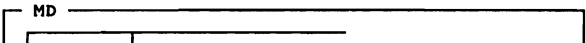
showing that the DICT section of the CLIENT file is to have a modulo of 1, and the data section is to have a modulo of 3. Since there will probably be fewer items on the DICT section of the file, this will normally have a smaller modulo than the data section.

The two parts of the file – the DICT and the data section – are physically quite separate sets of frames on disk; they may be quite close together or they may be some distance apart. When a file is created, both sections are completely empty, except for a pointer which is set up on the DICT; this is the CLIENT item which I've shown in Figure 4.9. Let me explain why this is needed.

When we use a command such as:

```
CREATE-FILE CLIENT 1 3
```

to create the CLIENT file, Advanced Pick will look around for a single unused frame which can be used for the DICT section (let's imagine that it allocates frame 54321 for this), and then it will look for a block of three contiguous frames for the data section (let's imagine that these start at frame 87654). It will then set up pointer items shown in Figure 4.10.



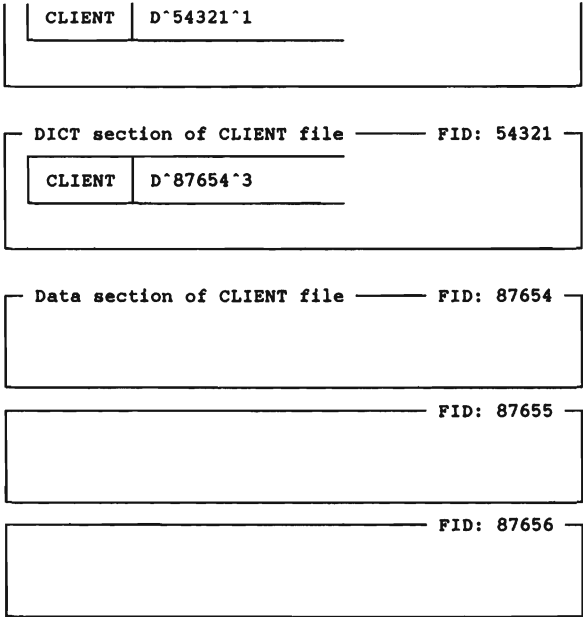


Figure 4.10: D-pointers for the CLIENT file

Because the first attribute of these items contains the letter D, these items are often called **D-pointers**.

The D-pointer on the MD has the item-id CLIENT and points to the DICT section of the file; the D-pointer on the DICT of the CLIENT file has the item-id CLIENT and points to the data section of the file. Each D-pointer holds the FID of the base frame and the modulo of the file to which it points. A simplified version of Figure 4.10 is shown in Figure 4.11; here, each large box represents a file, and each small, open box represents a D-pointer which identifies (points to) a file section.

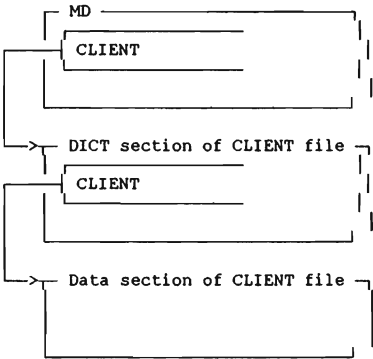


Figure 4.11: Linkage to find a file

The D-pointer on the MD which identifies the DICT section of a file is known as a **file definition** or a **file defining item**. The D-pointer on the DICT section of a file which identifies the data section of a file is known as a **data-level identifier** or a **file defining item**.

You may handle the information held on either section of a file. If I issue an Access sentence such as:

```
LIST DICT CLIENT
```

using the word **DICT** before the name of the file, this will produce a list of the items (including the data-level identifier and any Access attribute definitions) which are held on the **DICT** section of the **CLIENT** file, and if I issue a sentence such as:

```
LIST CLIENT
```

this will produce a list of the items (the data records) on the **data** section of the **CLIENT** file.

For most purposes, the **DICT** section of a file and the data section of the same file are two separate and distinct areas; the only connection between them is the logical linkage which is implied by the D-pointer.

4.2.2 DICT-only files

Not all your files will hold **STAFF** data or **CLIENT** data in which all the items have the same form; some files will contain miscellaneous collections of *unstructured* and *free-format* information such as text, documents, Procs and other information which it might not be appropriate to use with Access. If you did not wish to use the full facilities of Access with such a file, then you probably wouldn't need a separate dictionary section (to hold the Access dictionary definitions) and you could create the file with just one section, using a command such as:

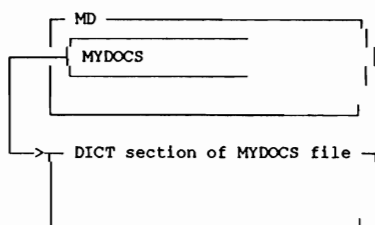


Figure 4.12: A DICT-only file

```
CREATE-FILE DICT MYDOCS 7
```

This would create a file organised like that in Figure 4.12. Note that the single section is considered to be a **DICT** section because, like all **DICT** sections, it *hangs* directly from the **MD**. I could then process the **MYDOCS** file with commands such as:

```
LIST DICT MYDOCS
```

or:

```
LIST MYDOCS
```

either of which would show me the contents of the **MYDOCS** file.

Such **DICT-only** files are not very common. The only advantage of having a **DICT-only** file is that you save the space which would be required for a dictionary which you would not use. I could have created the file in the normal manner:

```
CREATE-FILE MYDOCS 1 7
```

and then ignored the dictionary.

4.2.3 Files with more than one data section

Just as it is possible to have a file without a data section and files with a data section, it is also possible to have a file with several data sections.

This might be useful in situations where you have several distinct sets of data which must be kept separate but which can all use the same dictionary. For example, I once worked on a system for a client who needed to hold the data concerning the prices and other details of cars. Each month, the client received a diskette with the current prices for all models of new and used cars on the market. When he received February's prices, he wanted to hold these on the system but he still needed to retain January's prices.

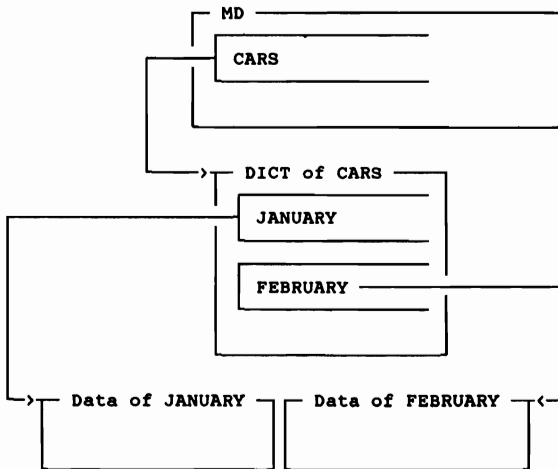


Figure 4.13: Several distinct files

It would have been perfectly realistic to create a new file to hold each month's figures, but – in order to avoid the maintenance problems of having two (or 12 or more) identical copies of the dictionary of the file – I decided to have a single file called CARS which had a separate data section to hold each month's figures. In this situation, the files were organised as in Figure 4.13. To create these files I first created the DICT-only CARS file:

```
CREATE-FILE DICT CARS 11
```

and then used a series of commands:

```
CREATE-FILE DATA CARS,JANUARY 117  
CREATE-FILE DATA CARS,FEBRUARY 117  
CREATE-FILE DATA CARS,MARCH 117
```

as each new monthly figures were loaded. The contents of the DICT could be handled by sentences of the form:

LIST DICT CARS

and the contents of any particular data section by references of the form:

CARS,month

in sentences such as:

LIST CARS,JANUARY PRICE MODEL WINDOWS DOORS
LIST CARS,FEBRUARY PRICE MODEL WINDOWS DOORS

and the definitions PRICE, MODEL, WINDOWS and DOORS would be placed on DICT of CARS where they were accessible to all the separate monthly files. To make life easier for the users, I also created a **file synonym** called CURRENT which was changed to refer to the latest file and could be used in sentences such as:

LIST CURRENT PRICE MODEL WINDOWS DOORS

4.3 Accounts

I have mentioned that there may be any number of files on your system. However, for security reasons, it is unwise to let every user have access to every file. Therefore, the files of any Advanced Pick system are grouped into **accounts**, and each account holds just the files that its users need. Thus, the STAFF file, the PAY-RATES file and the TAX.CODES file may be held by the WAGES account; the CLIENT file, the ORDER file, the PRICES file and the STOCK file may be held on the SALES account, and so on. To use the STAFF file, you must log on to the WAGES account, and to use the CLIENT file, you must log on to the SALES account.

Each account is identified by its MD, its master dictionary. When you log on to Advanced Pick, you will first be asked for your user-id (to make sure that you are an authorised user), and then you will be asked which MD you want to use; in other words, *which set of files you want to use*.

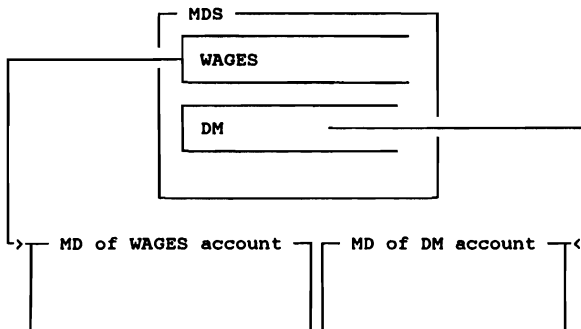


Figure 4.14: The MDS file and the MDs

The information about each account – that is, each master dictionary – is held on a file called MDS. For each account, the MDS file holds an item which has the account name as its item-id and which contains information about that account: the password, security codes, and – most important of all – the FID and the modulo of the master dictionary of the account. Figure 4.14 shows how we might visualise the MDS file and the way in which the individual items there point to each MD. There are just two accounts on this system: WAGES and DM.

The D-pointers on the MDS file which point to each MD are known as **account definitions** or **file defining items**. We shall have more to say about the structure of account definitions when we see how accounts are created and maintained.

4.3.1 Standard accounts

Anyone who has worked with R83 Pick will know that the system comes with a number of standard accounts: SYSPROG, ACC, BLOCK-CONVERT, SYSTEM-ERRORS and others. This is not the case on Advanced Pick which comes with just one account: DM, the Data Manager - account. This account owns all the standard files which were previously set up as separate accounts.

Some early versions of Advanced Pick came with three standard accounts: PA, the persal assistant account for simple administration tasks; QA, the quality assurance account for testing the system and reporting faults; TUTOR, a simple teaching account.

4.4 The hierarchy of files

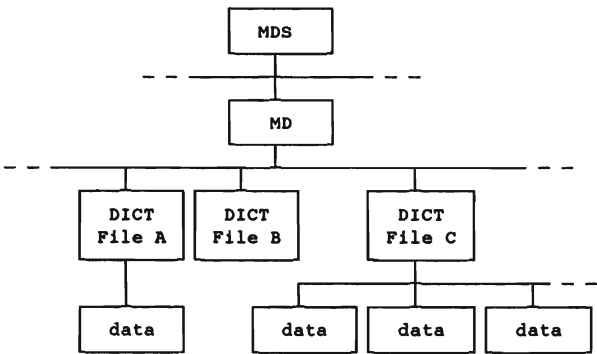


Figure 4.15: The hierarchy of files

The way in which the files are organised is of fundamental importance in understanding the logical and physical structure of your Advanced Pick system. In Figure 4.15, we summarise what we have learnt so far and

combine the diagrams from the previous Sections.

Each box in Figure 4.15 represents a **file**, and the lines connecting the boxes represent the linkage by which an item (a D-pointer) on one file enables Advanced Pick to locate a file *lower down* the diagram. Each D-pointer holds the base FID and the modulo of the file to which it points. The dotted links without boxes remind us that there may be any number of accounts (MDs) on the system, and any number of files on an account, and any number of data sections for a file.

The file called **MDS** is the most important file on any Pick system, and holds one item for each account on the system. This is a D-pointer and identifies the **MD**, the **master dictionary**, for that account.

The MD of each account holds:

- **File definitions** for all the files which have been created on that account.
- **File synonyms** and **Q-pointers** to all the standard files which support the system, and to files which are held on another MD but which are needed by this account.
- Definitions for all the standard **TCL verbs**. When you type a TCL command such as **OFF**, Advanced Pick looks for an item called **OFF** on your MD and then carries out the instructions held in that item.
- Definitions for all the standard **Access modifiers** and **connectives**,
- Definitions for any **macros** which are available to users of that account.
- Definitions for any **menus** which are available to users of that account.
- Pointers to all standard **Procs**, and to any Procs which you and other users have created.

Going further down the diagram, we find that the **DICT** section of each file contains a data-level identifier which points to the data section of that file. The **DICT** will also hold any definitions which have been created to enable you to use **Access** to produce reports on the data held on the file.

The data section of each file contains the true data items – our sales data, the personnel items, information about our clients, or invoice information.

A file may have one data section (as file A in Figure 4.15), it may be **DICT-only** file with no data section (file B), or it may have several data sections (file C).

There are some esoteric technical points which you might appreciate from Figure 4.15:

- A **D-pointer** – whether account definition or file definition or data-level identifier – only points to one file (one box on the diagram).

- Only the file where the D-pointer is held determines whether it is an account definition or a file definition or a data-level identifier.
- An **account** is any file for which there is a D-pointer on the MDS file.
- A **master dictionary** is any file for which there is a D-pointer on the MDS file.
- An **account** and its **master dictionary** are synonymous.
- A file always has a D-pointer on the master dictionary of the account to which it belongs.
- A file called PROGS held on the SALES account is quite separate from a file called PROGS held on the WAGES account.

D-pointers enable Advanced Pick to navigate from MDS to MD to DICT to data, and contain the following information about the file to which they point:

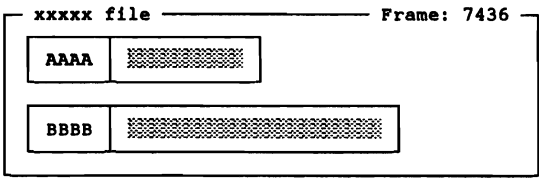
- The **base FID**, that is the frame identifier of the first frame in the file.
- The **modulo** of the file.
- The **security lock-codes** to control read and write actions on the file.
- The appearance of the **item-ids** on the file when they are shown on an Access report: the **left/right justification** and the **width** of the column when the item-id appears on an Access report.

STAFF		
01	D	<== Identifies a D-pointer
02	98765	<== Base FID
03	177	<== Modulo
04		
05		<== RET-LOCK lock-code
06		<== UPD-LOCK lock-code
07		
08		
09	L	<== Justification
10	10	<== Width

Figure 4.16: A D-pointer

A typical D-pointer is illustrated in Figure 4.16. Some D-pointers hold additional information, as we shall see when we look at them in more detail later.

Note that the USERS file is not an integral part of Advanced Pick structure in the way that the MDS file and the individuals MDs are. The USERS file is merely a data file belonging to the DM account which is used when a user logs on, and which holds identification and security information.



(a) The logical view

fid: 7436 : 0 0 0 0 (1DOC : 0 0 0 0)									
0000	00000000	53001000	41414141	FE546869	000	S...	AAAA^This:		
0010	73206973	20697465	6D204141	41412061	016	:s is item	AAAA a:		
0020	6E642069	74207769	6C6C2062	65206865	032	:nd it will be	he:		
0030	6C642064	69726563	746C7920	77697468	048	:ld directly with:			
0040	696E2074	68652066	696C6520	73706163	064	:in the file	spac:		
0050	652EFFFF	00007C03	15009000	42424242	080	:e.^_...	BBBB:		
0060	FE303030	384636FE	FFFFFFFF	FFFFFFFF	096	:^0008F6^	_____:		
0070	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	112	: _____:			

(b) The physical view: DUMP

Figure 4.17: Items on a file

4.5 Item format

Let us look more closely at the way in which items are actually held on disk. I have created a file with two items:

- 1) Item AAAA contains just one attribute: This is item AAAA and it will be held directly within the file space.
- 2) Item BBBB contains just one (very long) attribute: This is item BBBB and the data is 900 bytes long and, since it exceeds 847 bytes in length, it will be held indirectly, that is, with just a pointer in the file space. This pointer holds the frame address where the true data is to be found.

Figure 4.17 (a) shows the logical view of two items as I would have presented them in our earlier discussions; diagram (b) was produced by means of the DUMP command and shows the actual data frame where the items are held. The DUMP displays shown here were produced on an AP system which uses 1024-byte frames. We have shown only the first few bytes in the frame, the remainder of the frame is filled with padding and is not related to the data items. On the right of the DUMP display, you can see the characters held on the file, on the left, you can see their hexadecimal equivalents.

The line at the top of the display:

```
FID: 7436 : 0 0 0 0 ( 1D0C : 0 0 0 0 )
```

identifies this as frame 7436 (or 1D0C in hexadecimal notation). The zeroes mean that this frame is not linked to any other frame, either in front or behind. These are the **forward** and **backward** links that we met earlier; this linkage information is held in the final 24 (or 48) bytes of the frame (not at the front, as is the case on R83 Pick systems).

In the rest of the display, we see that, for each item, there is:

- An 8-byte **control field** which is held immediately before each item-id: this tells us the length of the item, how the item is held (that is, whether it is a direct item or an indirect item), and whether the item has been changed or deleted since the last file-save. For item AAAA, this control field is shown as s... (because some of the characters have no printed form, and are shown as dots; in hexadecimal, they appear as 0000000053001000).
- The **item-id**: this may be as long as necessary and is terminated by an attribute-mark. The attribute-mark is shown as ^ and FF in the two parts of the display.
- The **data** of the item for direct items, or the frame address for indirect items. Here we see the exact contents of item AAAA (a direct item), but we cannot see the contents of item BBBB (an indirect item).
- A final **attribute-mark** at the end of each item.
- The **end-of-item marker** (shown here as _ and FF) after each item, and, following the final item in the group, the **end-of-group marker** (shown here as _ and FF).

If we look more closely at the 8-byte control field as a hexadecimal string, we find that it is of the form:

```
00 00 00 00 yy xx zz 00
```

for item AAAA this is:

```
00 00 00 00 53 00 10 00
```

where xxyy is the hexadecimal length of the item (this represents the sum of the 8-byte control field itself, plus the length of the item-id, plus one attribute-mark, plus the length of the true data and field separators, plus 1 for the final attribute mark). Thus, item AAAA is 0053 bytes (that is, 83 decimal bytes) in length; note the way in which the xxyy field is held. The character zz is 00 or 10 or 01 and indicates that this is a **direct** item held in the file space; 00 denotes that the item has not changed since the last full file-save, 10 denotes that it has changed, and 01 would indicate that it had been deleted; we'll look at these codes later.

Items such as AAAA of which the data is 847 bytes in length or shorter are held in the file space (as on R83 Pick). This figure of 847 bytes refers to the length of data in the item and excludes the field-length count, the item-id and the final end-of-item marker. Longer items (848 bytes or

more), such as BBBB, are held **indirectly**, with the actual data being written to one or more frames of disk and only a pointer being held in the file space. On those systems which use 2048-byte frames, the size limit for direct items is 1696 bytes.

For indirect items, the 8-byte control field is of the form:

WW VV YY XX yy xx zz 00

for item BBBB, this is:

00 00 7C 03 15 00 90 00

where **WWVWXYZ** is the length of the true data item, **xyy** is the length of the pointer itself, the character **zz** being 80 (not changed since the last full file-save) or 90 (changed), or 01 (deleted), all implying that the item is held indirectly.

In Figure 4.17, we see that the pointer for item BBBB is 21 bytes long (= 0015 hexadecimal) and directs us to frame 2294 (= 8F6 hexadecimal) where we shall find an item which is 892 bytes long (= 0000037C hexadecimal), as indicated by the first part of the 8-byte control field held in the file space. Figure 4.18 shows frame 2294 to which the pointer for item BBBB directs, and we see the full 892 bytes of the item, up to and including the final attribute-mark.

```

fid: 2294 : 0 0 0 0 ( 8F6 : 0 0 0 0 )
000 :_BBBB^This is item BBBB and the data is 900 bytes :
050 :long and, since it exceeds 847 bytes in length, it:
100 : will be held indirectly, that is, with just a poi:
150 :nter in the file space. This pointer holds the fra:
200 :me address where the true data is to be found. XXX:
250 :XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX:
:
800 :XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX:
850 :XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX^_____:
900 :_____:
950 :_____:

```

Figure 4.18: The actual indirect item

File definitions and data-level identifiers - D-pointers - are held in this indirect manner, with the character **zz** being A0 (not changed), B0 (changed), or 21 (deleted). Object programs are held in this indirect manner, with the character **zz** being C0 (not changed), D0 (changed) or 01 (deleted).

After a full file-save has been performed, the system resets the character

indicators to the clean states of 00, 80, A0 and C0 during the process of clearing the dirty bits.

The Advanced Pick literature also uses the terms **in-line item** where I have used *direct item*, and **out-of-line item** where I have used *indirect item*.

4.5.1 Deleting items

Because the **incremental file-save** feature of Advanced Pick needs to know what changes have been made to a file since the last file-save, when an item is deleted from a file, there must be some record of that fact. This is achieved by retaining the item-id on the file and setting the dirty bits to indicate that the item has been deleted. Thus, if we had deleted, say, an item with the item-id D we would find that the file space still held an item:

```
.....D^_
```

or, in hexadecimal:

```
000000000B00010044FEFFFF
```

that is, an **empty-pointer** still remains, and the dirty bits show that the item has been deleted. If we list the file, item D will not appear and if we create a new item D this empty-pointer will be re-used and the dirty bits will show that the item has been changed since the last file-save.

4.6 Working with files

We have looked closely at the physical and logical aspects of files on Advanced Pick. Now, we shall look at the practical aspects of using and working with files.

4.6.1 Creating files

Before creating a file, the Systems Analyst will have carried out an investigation and produced some estimate of the size of the items and the number of items which the file will hold, and used these to determine a suitable modulo for the file. We discuss this later.

A file is created by means of the **CREATE-FILE** command. The command has several forms:

```
CREATE-FILE PAYROLL 3 11
```

to create a file called PAYROLL which has a modulo of 3 for the DICT section and a modulo of 11 for the data section.

```
CREATE-FILE DICT BROCHURE 11
```

to create a DICT-only file called BROCHURE which has a modulo of 11.

```
CREATE-FILE DATA CARS,DECEMBER 133
```

to create new data section on the DICT of the CARS file; the new data section is called DECEMBER and has a modulo of 133.

It is possible to specify options on the CREATE-FILE command.

```
CREATE-FILE PAYROLL 3 11 (L
CREATE-FILE DICT BROCHURE 11 (P
CREATE-FILE DATA CARS,DECEMBER 133 (LS
CREATE-FILE WORKER 1 3077 (X
```

In all cases, these add extra letters to the D in all the D-pointers which are established by the command, and this affects the way in which the file is handled. The available options are L (to indicate that any changes made to the file are to be logged by the transaction logger), P (to indicate that the primary file space is to contain only pointer (indirect) items regardless of the size of the items), S (to indicate that the item-ids on the file are to be case-sensitive), X (to indicate that the file is to be ignored during file-save / account-saves), and/or Y (to indicate that the contents of the file are to be ignored during file-save and account-saves). We met these earlier.

If you wish to change any fields in the D-pointers after you have created the file, you can do this by means of the commands:

```
UD MD PAYROLL
    to change the file definition on the MD, or
```

```
UD PAYROLL PAYROLL
    to change the data-level identifier on the DICT of the file.
```

In general, D-pointers are rarely changed in this manner, although the System Manager may wish to set the lock-codes in attributes 5 and 6. The codes in attribute 1 of a D-pointer may also be set by the SET-DPTR verb.

4.6.2 LISTFILES: listing files

The LISTFILES command is used to produce a list of all the files which belong to an account and all the file synonyms which are available to the account. The command has the form:

```
LISTFILES
```

to display the report such as that in Figure 4.19 (a), or:

```
LISTFILES (P
```

to print the same report. The report shows the name, base FID and modulo of the files, and the entries are grouped by the CODE: D-pointers coming first, then DL-pointers, and so on, with Q-pointers (the file synonyms) coming last). If the (C option is used:

```
LISTFILES (C
LISTFILES (PC
```


only the names of the various D-pointers will be shown in a four-column report.

Page 1 *** DICT DM,MD, *** File Defining Items 29 Jul 1994

DICT DM,MD,. CODE F/BASE... F/MOD. DESCRIPTION.....

ABS	D	3348	1
ACCOUNTS	D	8499	1 SYSTEM USAGE ACCOUNTING
:			
POINTER-FILE	D	8573	1 STORES DATA FROM SAVE-LIST VERB
TCL-STACK	D	5957	1 TCL COMMAND STACK
USERS	D	8593	1 AUTHORIZED USERS
WORDS	D	5999	1

JOBS	DP	8353	1 PHANTOM JOBS STATUS AND HISTORY

ACC	Q		DACCOUNTS SYSTEM USAGE ACCOUNTING
ADMIN.BP	Q		BP.QA,ADMIN

:

STAT-FILE	Q		FILE-OF-FILES MASTER FILE OF FILES
SYSPROG-BP	Q	DM	BP SYSTEM UTILITY PROGRAMS
SYSPROG-PL	Q	DM	BP
SYSTEM	Q	MDS	HIGHEST LEVEL FILE
SYSTEM-ERRORSQ		DM	DM ERRORS
TS	Q		TCL-STACK TCL COMMAND STACK

**

Figure 4.19: LISTFILES

Other forms of the LISTFILES command are available:

LISTFILES CARS

to produce a list of all the data sections which are dependent upon the DICT section of the CARS file.

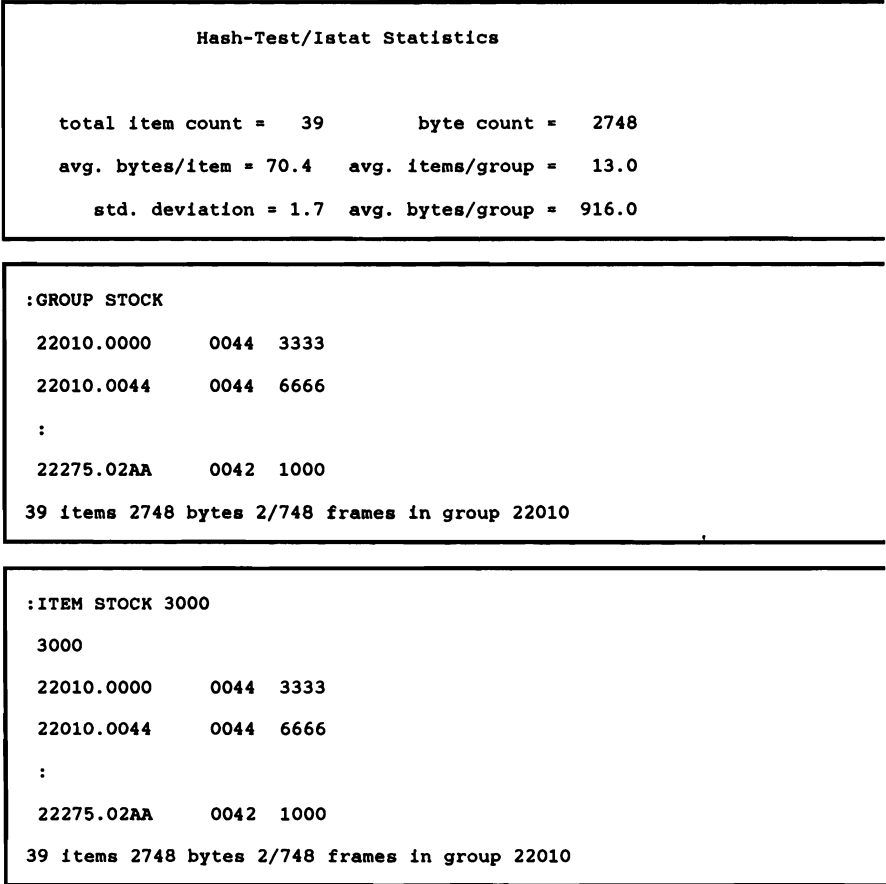


Figure 4.20: ISTAT / HASH-TEST / GROUP / ITEM

ISTAT STOCK
display the file statistics and a bar chart showing the distribution of the items within the groups of the file. Here, we see that the file has a modulo of 1 so there is only one bar of 39 > signs, one > for each item on the file.

HASH-TEST STOCK
display the same information as ISTAT but using a hypothetical modulo entered by the user in response to the TEST MODULO prompt. This is useful for assessing the impact of changing the modulo of a file. Here, we see the bar chart showing the distribution of the 39 items if we had created the file with a modulo of 3. The HASH-TEST command does not change the physical file in any way but simply carries out a what-if investigation.

GROUP STOCK
display the details of the items in each group of the file. Here, we see that the file starts in frame 20010 and the first (and only) group

contains 39 items occupying 2748 bytes, that is, 2 full frames and 748 bytes of a third frame.

ITEM STOCK 1000

display the details of the items in the group which contains (or should contain) item 1000.

4.6.4 Clearing files

Once the Systems Analyst has created a file, the data items will be added / maintained / deleted by means of application programs. We'll look at this in a moment.

It is possible to delete *all* the items on a file by one of the forms of the CLEAR-FILE command:

CLEAR-FILE STAFF

CLEAR-FILE DATA STAFF

either of these commands will delete all the items from the data section of the STAFF file. The contents of the DICT section are not affected by this command.

CLEAR-FILE DICT STAFF

will delete all the items (typically, these will be the attribute definitions for the STAFF file) from the DICT section of the STAFF file. Any D-pointers on the DICT section are not affected by this command, nor are the contents of the data section of the file.

CLEAR-FILE CARS, JULY

will delete all the items from the JULY data section of the CARS file. The DICT section of the CARS file and any other data sections are not affected by this command.

You might need to clear a file in this manner so as to get rid of all the old data prior to copying items from another file or before restoring data from a diskette/tape. In practice, you may find that you seldom use the CLEAR-FILE command.

If you use any of the forms of the CLEAR-FILE command shown above, Advanced Pick places a segment-mark at the front of the first frame in each group; the data can still be inspected by means of the DUMP command but it is otherwise gone and inaccessible. If you clear a file with the command:

CLEAR-FILE TESTD (C

the C option **scrubs** the frames occupied by the file, filling them with segment-marks, thereby obliterating all traces of the data which was held there.

In all cases, when a file is cleared, any overflow frames which were linked on to the primary space will be released and returned to the pool of unused frames.

4.6.5 Deleting files

When a file is no longer required, it may be removed from the system and the frames which it occupied returned to the pool of unused frames. This is done by one of the forms of the DELETE-FILE command:

DELETE-FILE MYDOCS
to delete both the DICT section and the data section of the file MYDOCS. The contents of both sections will be lost.

DELETE-FILE DATA CARS, JANUARY
to delete the data section JANUARY from the DICT of the CARS file. The DICT section itself and all other data sections (other than JANUARY) are not affected by this command.

DELETE-FILE CARS
to delete the DICT section of the CARS file and *all* the data sections.

Once an application system has stabilised, it is unlikely that much use will be made of the CREATE-FILE / CLEAR-FILE / DELETE-FILE commands.

4.6.6 The FILE-OF-FILES and file changes

The MDS file and USERS file are important to the running of your Advanced Pick system. Another important file is the FILE-OF-FILES file.

Whenever a file is created or restored from backing storage, a new item is created on the FILE-OF-FILES file. Subsequently, whenever a file is cleared / deleted / renamed / restored from backing storage, the appropriate FILE-OF-FILES item is updated to record the activity, the date and time when it was performed, and the user-id and the port-number which carried out the action. These items hold a variety of information about each file on the system, as we shall see when we discuss file statistics and the FILE-OF-FILES file later.

4.6.7 D-pointers

The file definitions and data-level identifiers can contain much more information than the simple items we saw earlier. The full form of D-pointers is illustrated in Figure 4.21.

- 1) Attribute 1: This identifies the item as a file defining item, defining the DICT section or the data section of a file. This attribute consists of the letter D followed by a suitable combination of the following codes:

01	dictionary-code
02	base
03	modulo
04	not used
05	retrieval-lock
06	update-lock
07	not used
08	correlative
09	justification
10	column-width
11	not used
12	not used
13	reallocation
14	input-conversion
15	macro
16	output-macro
17	description
18	not used
19	not used
20	hotkey.all
21	hotkey1
:	
29	hotkey9
30	hotkey0

Figure 4.21: D-pointers

- L indicates that this file is to be the subject of **transaction logging**, and all changes to this file are to be logged. This is not used with the X or Y options.
 - P indicates that the all items on the file are to be held as **indirect items**. We have seen that short items are normally held directly in the file space and longer items are held indirectly with only a pointer in the file space. As we can deduce from our earlier discussions, organising a file in this manner, means that the process of finding any particular item is much faster, since the physical items on the file are small, but the retrieval of any item will always require at least two disk read operations (one or more to find the pointer, then a further one or more to retrieve the item from the frames of virtual memory).
 - S indicates that the item-ids are to be **case-sensitive**. Thus, on a file defined with this code, the item-ids ABC/123, Abc/123 and abc/123 would identify three distinct items.
 - X indicates that this file is to be ignored completely when a file-save or an account-save is carried out. After a restore, the file will no longer exist.
 - Y indicates that the contents of this file are to be ignored when a file-save / account-save save is carried out. After a restore, only an empty file will be exist.
- 2) Attribute 2: The **base FID**, that is the FID of the first frame of the primary space allocated for the file.
 - 3) Attribute 3: The **modulo** of the file.
 - 4) Attribute 4: This is null. R83 Pick systems used this to hold the separation of the file, but the concept of separation has been discontinued on AP.
 - 5) Attribute 5: The **RET-LOCK lock-code** of the file. This code must match the equivalent KEYS attribute of the user definition if that user is to be allowed to read from the file.
 - 6) Attribute 6: The **UPD-LOCK lock-code** of the file. This code must match the equivalent KEYS attribute of the user definition if that user is to be allowed to write to the file.
 - 7) Attribute 7: This is not used.
 - 8) Attribute 8: This is normally null.
 - 9) Attribute 9: The **format and justification** of the item-ids of the file.
 - 10) Attribute 10: The **column width** for the item-ids of the file.
 - 13) Attribute 13: This is the file **reallocation** parameter. If the System

Manager were to change this attribute to:

(117)

this would indicate that, when this file is next restored during a file-restore or an account-restore operation, it is to have a modulo of 117. This value will then appear in attribute 3.

14) Attribute 14: This specifies the **input-conversion**.

15) Attribute 15: This specifies the **macro**, a list of datanames which are to be used when zooming to another item with the Update Processor.

16) Attribute 16: This specifies the **output-macro**, a list of datanames which are to be shown if this file is used without an output list of fields in an Access report.

17) Attribute 17: This is the **description**, a free-format text field for any comments concerning the usage and nature of the file.

20) Attributes 20 onwards: These contain the CALLs to any subroutines which are to be invoked by the **hotkey** sequences.

All fields of D-pointer items (except attributes 2 and 3) may be set and/or changed by the System Manager and/or the Systems Analyst, as we shall see in the following section.

We shall look more closely at attributes 9, 14, 15, 16 and 17 later in our discussion of attribute definitions. We discuss hotkeys in Chapter 10.

4.6.8 UD and D-pointers

The D-pointers for a file are established when the file is created. Subsequently, the System Manager or the Systems Analyst may wish to make certain changes to a D-pointer ...

- o to change the file reallocation parameter,
- o to change the macro and/or output macro specifications,
- o to change the security lock-codes,
- o to change the hotkey routine names,

and so on. You will have realised that – no matter whether it is an account definition, a file definition or a data-level identifier – every D-pointer is an important part of the structure of the Advanced Pick database. If you look again at Figure 4.15 you will see that, if one of the links there was broken, damaged or lost, then all the files below that link would be lost. To minimise such risks, Advanced Pick will not let you handle D-pointers as you handle other simpler items; you cannot edit a D-pointer, you cannot copy a D-pointer. Any changes to a D-pointer must be made by means of the UD macro:

to change the account definition D-pointer which defines the MD of the WAGES account and is held on the MDS file.

UD MD STAFF

to change file definition D-pointer which defines the DICT section of the STAFF file and is held on the MD.

UD STAFF STAFF

to change the data-level identifier D-pointer which defines the data section of the STAFF file and is held on the DICT of the STAFF file.

UD CARS DECEMBER

to change the data-level identifier D-pointer which defines the DECEMBER data section on the DICT of the CARS file.

The UD macro allows you to change certain parts of the D-pointer, but not others (such as the base FID or the modulo); and of those fields which are changed, the macro checks that your changes are consistent and will not upset the vital linkage which the D-pointer represents.

4.6.9 SET-DPTR

If you simply want to amend attribute 1 of an account definition or the D-pointers for a file, possibly changing a D-pointer to a DL-pointer, or a DX to DY, then you can use the SET-DPTR command. This has the general form:

```
SET-DPTR {+xxxx} {-xxxx} dddd {(options)}
```

to add (+) or remove (-) the letter (or letters) xxxx to the D-pointer for dddd. The string of letters xxxx will be one or more of X and Y and L and P and S.

The options include:

- A** to change an **account definition**.
- P** to print a record of the changes.
- Q** to suppress the messages confirming the action.

SET-DPTR +P STOCK

to add the letter P to the D-pointers for the DICT section and the data section(s) of the STOCK file.

SET-DPTR +L ARCHIVE

to add the letter L to the D-pointers for the DICT section and the data section(s) of the ARCHIVE file.

SET-DPTR +Y CARS,DECEMBER

to add the letter Y to the data-level identifier for the DECEMBER data section of the CARS file on this account.

SET-DPTR +X SALES,CARS,DECEMBER

to add the letter X to the data-level identifier for the DECEMBER data section of the CARS file on the SALES account.

SET-DPTR +X WAGES,PAY-RATES,

to add the letter X to the D-pointers for the DICT section and the data section(s) of the PAY-RATES file on the WAGES account.

SET-DPTR +X -L WAGES,PAY-RATES,

to add the letter X and remove the letter L from the D-pointers for the DICT section and the data section(s) of the PAY-RATES file on the WAGES account.

SET-DPTR +X SALES,CARS,DECEMBER

to add the letter X to the data-level identifier for the DECEMBER data section of the CARS file on the SALES account.

SET-DPTR +X WAGES (A

to add the letter X to the account definition for the WAGES account.

SELECT MDS WITH A1 = "[X]" "[Y]"

SET-DPTR -XY (AP

to remove X and Y from all account definitions on the MDS file, and

print a list of the changes.

SET-DPTR STOCK

to display (without changing) the D-pointers for the DICT and all data sections of the STOCK file.

The SET-DPTR process only considers attribute 1 of the D-pointers and displays these before and after the changes. If the D-pointer(s) already include letters which are being added and/or do not include letters which are being removed, then no action is taken. Items which are not D-pointers are ignored.

Currently, it is not possible to change just the D-pointer for the DICT section of a file. However, this can be achieved by first changing all the D-pointers for the file, and then undoing the changes to the data section:

```
SET-DPTR +X WAGES
SET-DPTR -X WAGES,WAGES
```

Some forms of SET-DPTR are equivalent to the :SWX, :SWY and other verbs of R83 Pick.

4.6.10 MOVE-FILE: moving D-pointers

We have already noted that it is only the location of a D-pointer which determines whether it is an account definition, a file definition and a data-level identifier. It is possible to move a D-pointer from one file to another. Because of the integrity checks which must be imposed by Advanced Pick, this cannot be done by means of the ordinary COPY or editing processes; instead, you will use the special MOVE-FILE verb which moves a D-pointer from one file to another. The command is similar in form to the COPY command:

```
MOVE-FILE md.name file.name
```

where *md.name* is the name of the file (typically, the MD) which holds the D-pointer to the file *file.name*, and *file.name* is the name of the D-pointer which is to be moved. The process will then ask for the destination of the D-pointer:

TO:

to which you will enter the destination response. This will have one of the forms:

```
(dest.md.name dest.file.name
OR:
(dest.md.name
OR:
dest.file.name
```

where *dest.md.name* is the name of the file to which the D-pointer is to be moved, and *dest.file.name* is the name by which the file D-pointer is to be saved. In all cases, the D-pointer will be removed from its original

location and moved to the new location. In none of these instances will any Q-pointers which refer to the old and/or the new files be set up or changed. Here are some complete examples:

MOVE-FILE MD STAFF
TO: (DM,,
to move the D-pointer for the (DICT of the) STAFF file from the MD of the current account to the MD of the DM account, thereby making DM the owner of the STAFF file.

MOVE-FILE DM,, WORKER
TO: (MD
to move the D-pointer for the (DICT of the) WORKER file from the MD of the DM account to the current account, thereby making the current account the owner of the WORKER file.

MOVE-FILE MD STAFF
TO: NEWSTAFF
to move the D-pointer for the (DICT of the) STAFF file on the MD of the current account and saved it with the name NEWSTAFF, thereby changing the name of the STAFF file to NEWSTAFF.

MOVE-FILE DICT STAFF STAFF
TO: NEWSTAFF
to move the D-pointer for the (data section of the) STAFF file which is held on the DICT section of the STAFF file and save it with the name NEWSTAFF, thereby changing the name of the data section of the STAFF file to NEWSTAFF.

Early versions of Advanced Pick had two additional verbs:

STEAL-FILE
RENAME-FILE

but the features of these have been assimilated by MOVE-FILE.

4.6.11 Q-pointers / file synonyms

In the simplest situation, a file which belongs to one account is inaccessible to users working on another account. This means that, if you are working on the WAGES account and you create a file called, say, PAY-RATE, then the D-pointer which identifies the file – shown in Figure 4.22 (a) – is created on your WAGES MD and known only to you; other accounts do not have any means of reaching the D-pointers and cannot therefore reach your PAY-RATE file.

	MD of WAGES account	MD of LEDGER account	MD
00	PAY-RATE	XXXXX	QFILE
01	D	Q	Q
02	76543	WAGES	DEVELOPMENT
03	3	PAY-RATE	PROGS
04			
05			
06			
07			
08	L		
09	10		

(a)

(b)

(c)

Figure 4.22: D-pointers / Q-pointers

However, there may be occasions when a user on one account needs to be able to use the data held on a file owned by another account, or when users working on different accounts need to share the same files. For example, the DM account owns a number of general-purpose files which are of use to all accounts; it would be wasteful (and difficult to maintain) if every account had its own copies of these files, so each account has a set of pointers which direct the user to the DM files. The situation is resolved by the creation and use of **Q-pointers** or **file synonyms**, and, as we shall see later, by the use of **pathnames**. A Q-pointer is an item on the MD of one account which points to a file belonging to another account. A typical Q-pointer is shown in Figure 4.22 (b); the item XXXXX is held on the MD of the LEDGER account and refers to the file called PAY-RATE which belongs to the WAGES account. Thus, someone who is logged to the LEDGER account could issue TCL commands and Access sentences such as:

```
COUNT XXXXX
EDIT XXXXX *
LIST XXXXX WITH CODE = "L]"
UPDATE XXXXX 'L123' CODE
UD XXXXX CODE
```

and these commands would actually process the PAY-RATE file which belongs to the WAGES account.

You can create your own Q-pointers by using the EDITor or the Update Processor to set up an item – like that in Figure 4.22 (b) – on your MD; such a Q-pointer may have any item-id where I used XXXXX. The format of the item is as shown here: attribute 1 contains just the letter Q, attribute 2 contains the name of the account, and attribute 3 contains the name of the file on that account. If attribute 2 is null, the current account is assumed, and if attribute 3 is null, the MD of the account is assumed.

It is possible to set up a Q-pointer to point to a file on your own account. You might do this so that a file can be called by more than one name; for example, during development work, you might have created a file called INVENTORY.DEV.1200A and carried this over to the live system, but you might prefer to provide a Q-pointer with a more friendly name – such as CATALOGUE – for your users.

The SET-FILE command will set up a Q-pointer to a file. The forms of the command are:

```
SET-FILE DEVELOPMENT PROGS
```

to create, on the MD of the current account, a Q-pointer called QFILE which will refer to the PROGS file owned by the DEVELOPMENT account, as illustrated in Figure 4.22 (c).

```
SET-FILE DEVELOPMENT PROGS (S
```

to create, on the MD of the current account, a Q-pointer called QFILE

and then copy this to create a new Q-pointer called PROGS (the original file name) which will refer to the PROGS file owned by the DEVELOPMENT account. If PROGS already exists on this MD, then the new PROGS Q-pointer will not be created.

SET-FILE DEVELOPMENT PROGS (OS

to create, on the MD of the current account, a Q-pointer called QFILE and then copy this to create a new Q-pointer called PROGS (the original file name) which will refer to the PROGS file owned by the DEVELOPMENT account. If an item PROGS already exists on your MD, then the new PROGS Q-pointer will overwrite the original item (unless that is a D-pointer).

Note that the item QFILE is only intended to serve as a temporary name, and unless the extended form of the command is used, each use of the SET-FILE command will create a new version of the QFILE Q-pointer pointing to a different file. This can be a source of error, danger and confusion if several users of an account are using SET-FILE at the same time. A solution can be found in using the Update Processor or the EDITor to create your own Q-pointers with distinct names, or use the extended form of the SET-FILE command which allows you to specify a name by which the Q-pointer is to be saved:

SET-FILE WAGES INVENTORY.DEV.1200A CATALOGUE

which would create a Q-pointer called CATALOGUE, allowing the INVENTORY.DEV.1200A file to be identified by a more memorable name.

You may have several different Q-pointers all referring to the same file, and you may have a Q-pointer which points to another Q-pointer. This is convenient if you have set a Q-pointer to a file and the owner later moves the file to another account and replaces the D-pointer by another Q-pointer. R83 Pick users may remember that this was not possible on earlier systems.

The ability to use Q-pointers - and, more importantly, to have Q-pointers pointing to Q-pointers - means that users need no longer be aware of where their files are physically held. This concept of **data transparency** is an important characteristic of database theory.

Whenever you try to use a Q-pointer to reach someone else's files, the access path will be checked and any non-matching lock-codes will cause the process to terminate.

4.6.12 Pathnames

A much simpler means of reaching a file on another account is by using a **pathname**. Pathnames can be used in any context where a file name is expected, and they have any of these forms:

WAGES, ,

to identify the MD of the account WAGES.

WAGES, PAY-RATES,

to identify the data section of the file PAY-RATES which belongs to the account WAGES, or the equivalent ...

WAGES, PAY-RATES, PAY-RATES

to identify the data section of the file PAY-RATES which belongs to the account WAGES.

VSALES, CARS, DECEMBER

to identify the data section DECEMBER of the file CARS which belongs to the account VSALES.

Note that there must always be two commas, and the trailing comma(s) are essential in the first two cases.

Some typical uses of pathnames in TCL commands and Access sentences might be:

```
EDIT SYSPROG, , CREATE-ACCOUNT
COPY TRAINING, STOCK,
UPDATE PERSONNEL, STAFF, RETIRED 1234/AA
EDIT DICT TRAINING, STOCK, DESCRIPTION
UD TRAINING, STOCK, QUANTITY MATERIAL
LIST WAREHOUSE, STOCK, DESCR TOTAL VALUE
```

Pathnames can also be used in Basic statements such as:

```
OPEN 'WAREHOUSE, STOCK, ' TO STOCK.FV ELSE STOP
```

As with Q-pointers, the access path to the file will be checked and if any non-matching lock-codes are encountered, the process will terminate.

4.7 Standard files

In addition to the MDS file and the USERS file, there are a number of other files which support Advanced Pick and which are available to all accounts. We shall discuss many of these later in the book. The list of such standard files includes:

- **ABS:** holds information about the executable object code used by the Advanced Pick system.
- **ACCOUNTS:** holds a history of the log on and log off dates and times, the connection time, and the computer usage for each user. We shall look at the ACCOUNTS file in a moment.
- **BLOCK-CONVERT:** holds the specifications for the large characters used by the BLOCK-PRINT verb.
- **BP:** holds the Basic programs for a number of system utilities.
- **DEVICES:** holds an item for every terminal and printer device recognised by the system. An item on this file is used to control all

cursor positioning and special effects at the user's terminal, and other items on this file are used to control the printer characteristics to each form-queue. A number of standard terminal / printer definitions are provided, and new items may be set up by the **DEFINE-TERMINAL** command.

- **ERRORS**: holds any system error messages reported by AP. The System Manager will inspect this file from time to time in order to monitor how the system is functioning. There is a separate data section – **ERRORS,GFE** – which is used to hold dumps of any frames in which GFEs are encountered.
- **FILE-OF-FILES**: holds statistical information for each file on the system.
- **FUNCKEYS / IOMAP-FILE / KB.FK / KB.PC / KEYBOARDS**: hold items used by the commands which set up the keyboard and the function keys.
- **JOBS**: holds details of each job submitted to the phantom scheduler.
- **MESSAGES**: holds the text of the error- and warning-messages displayed by the system processes.
- **NEWAC**: holds all the master dictionary items which are copied to the MD of each new Advanced Pick account created by the **CREATE-ACCOUNT** process.
- **PIBS**: holds a text description of the location and the device specifications for each port.
- **POINTER-FILE**: holds lists saved by the **SAVE-LIST** command.
- **TCL-STACK**: holds the stacks of all TCL commands issued by the users, and is used by the TCL stacker utility.

Let's look at some of these files more closely.

4.7.1 ACCOUNTS

ACCOUNTS is a system-wide file which holds a history of the dates and times when each user logged on and off the system, the connection time, the computer usage during the session, and the number of pages printed. Only those users who have A in attribute 9, the **OPTIONS** field, of their item on the **USERS** file will be logged on the **ACCOUNTS**; other users will be ignored.

The System Manager will produce reports of the accounting information by means of the:

```
LISTACC
or:
LISTACC (P
command.
```

The ACCOUNTS file should be cleared down from time to time.

Some organisations use the ACCOUNTS information as a basis for recharging for computer usage. When a new user is created, the default is *not* to log the accounting information; the accounting information for a particular user may be initiated by setting A in the OPTIONS field of the USERS item for that user, and may be suppressed by removing the A option. Accounting for all users may be suppressed by removing the ACCOUNTS file from the system.

The file has a synonym ACC which will be familiar to R83 Pick users.

4.7.2 FILE-OF-FILES

FILE-OF-FILES is a most important file and holds information about each file on the system. The FILE-OF-FILES file is created when a file-restore is performed, and new items are added whenever a file is created or restored to the system by an account-restore. Whenever a file is created / cleared / restored / deleted, the appropriate item on the FILE-OF-FILES file is updated. The statistics about the number of items and frames for each file are updated when a file-save is performed.

The following datanames are amongst those which are available when producing reports on the FILE-OF-FILES file:

- **FILE#** the sequential item.id assigned in the order in which the file was created / restored to the system.
- **MD** name of the MD (account) owning the file.
- **FILE-NAME** name of the file in MD of the account.
- **DATA-NAME** name of the data section.
- **FNAME** name of the file and the data section.
- **NAME** name of the account, the file and the data section.
- **BASE** base FID of the file.
- **MOD** modulo of the file.
- **ITMS** number of items in the file.
- **PTR-ITEMS** number of indirect items saved.
- **OVF-ITMS** number of item which were wholly or partially held in the overflow space.
- **BYTES** total number of bytes in the file.
- **PTR-BYTES** number of bytes in all indirect items.
- **F-SIZE** total number of bytes in the primary file space.
- **FRAMES** total number of frames in the file.
- **P-FMS** number of frames of indirect items.

- **F-FMS** total number of frames in the primary file space.
- **T-FRAMES** total number of frames in the file space.
- **SVDATE** date when file was last saved.
- **REEL#** volume number of the diskette/tape where this file was saved.
- **SEQ#** sequence number in which the file was saved.
- **OPENDATE** date when file was last opened.
- **STAT#** number of the last file-save at which this file was saved.
- **FILE-CODE** file status code indicating actions performed on the file, This is a multivalued sequence of: C clear file; D delete file; N new file; R rename file; T restored from backing storage.
- **TIMEDATE** date and time when the activity indicated by FILE-CODE occurred.
- **USER** user-id and port-number where the activity in FILECODE was performed .
- **DX/DY-DATE** date when DX/DY file was skipped during the file-save.
- **SAVE-LIST** attribute 25 onwards contains a list of items to be saved for this file, or * to save all items.

R83 Pick users will recognise that the FILE-OF-FILES file has many of the characteristics and uses of the STAT-FILE.

4.7.3 MESSAGES

MESSAGES is a system-wide file and holds definitions for all the error- and warning-messages which are displayed by the system processes. The messages held here may also be output directly by the Basic STOP statement, the ERROR statement and by the PRINT-ERR command.

The messages are not all *error*-messages, many are warning messages and general information messages:

- Successful completion messages:

```
FILE 'STAFF' DELETED
```

- Confirmation and information messages:

```
FILE 'STAFF' CREATED; BASE = 7001, MODULO = 1
```

- Basic compiler messages (both errors and information messages) such as:

```
PROGRAM 'xxxxx' COMPILED. 10 FRAME/S USED.  
COMPILATION ABORTED; NO OBJECT CODE PRODUCED.
```

- Basic execution messages, produced as a Basic program is being executed, including non-fatal warning-messages such as:

VARIABLE HAS NOT BEEN ASSIGNED A VALUE; ZERO USED!
DIVIDE BY ZERO ILLEGAL; ZERO USED!

and fatal error-messages, such as:

ARRAY SUBSCRIPT OUT-OF-RANGE
FILE HAS NOT BEEN OPENED

The **MESSAGES** file is used as a repository for several other items, including:

- The ***BLIST** item which controls the indentation used by the **B/LIST** utility.
- The **LOGON** message which is displayed immediately after logging to an account and before passing control to the logon procedure for that account.
- The **LEGEND** item is an Output Processor item which is shown at the foot of each printed report produced by the Access processor.
- The **SEQ** item which controls the sort sequence used by the MS processing code.
- The **VERSION** item which holds details of the current version of your Advanced Pick system.

The definition for each message is held as an item on the **MESSAGES** file, and each item is identified by its *message-id*. Each item consists of a sequence of instructions, one instruction per attribute, and these are used to build a string of data for the final message which is to be displayed at the terminal.

535
01 E Line specified is unavailable - try another.

(a) Standard **MESSAGES** item 535

MYERR1
01 HThe file ' 02 A 03 H' cannot be opened

(b) A user-written message: one parameter

Figure 4.23: **MESSAGES**

As an example of a simple error-message, we might look at **MESSAGES** item 535 shown in Figure 4.23 (a). This particular message is used by the

LOGOFF command if you attempt to log off a non-existent port. The MESSAGES item contains a single instruction to the message processor:

```
E Line specified is unavailable - try another.
```

here the E indicates that the output is to include the message-id (enclosed in brackets) followed by the text of the message. If this item were used by the LOGOFF process, or, for demonstration purposes, in a TCL command such as:

```
PRINT-ERR MESSAGES 535
```

or in a Basic statement such as:

```
READ RECORD FROM ITEM.ID ELSE STOP 535
```

```
OR:  
ERROR 535
```

then the display would be:

```
[535] Line specified is unavailable - try another.
```

A Systems Analyst may wish to create his/her own messages on the MESSAGES file and incorporate these into an application system. For example, he/she might create an item like that in Figure 4.23 (b) and save it on the MESSAGES file. In this example, the H instructions build an output string, and the A instruction incorporates a parameter passed into the message; if our Basic program used this message in a statement such as:

```
OPEN 'STOCK' ELSE STOP MYERR1, 'STOCK'
```

then the word STOCK would be slotted into the text of the message (at the A instruction) and the program would display the message:

```
The file 'STOCK' cannot be opened
```

if the program failed to open the STOCK file.

A	output a parameter.
A(n)	output a parameter left-justified in a field of <i>n</i> spaces.
B	sound the bell on the terminal.
C	clear the screen.
D	output the date in the format <i>dd mmm yyyy</i> .
E	output the message-id enclosed in square brackets.
Exxx	output the message-id, enclosed in square brackets and followed

by the string *xxx*.

Hxxx output the string *xxx*.

L start a new line.

L(n) skip *n* lines.

R(n) output a parameter right-justified in a field of *n* spaces.

S(n) skip to position *n+1* of the line.

T output the time in the format *hh:mm:ss*.

x skip a parameter.

Figure 4.24: Codes used in MESSAGES items

Figure 4.24 shows the codes which are used in constructing MESSAGES items. One of these codes must appear in the first position of each instruction of the message item. The items are created manually by means of the Update Processor or the Editor.

4.8 Working with items

In most situations, the items on your data files will be created, changed, inspected, reported and deleted by means of a suite of applications programs. For example, when you wish to add the details of a new employee on the STAFF file, you will probably run the specially-written ADD.NEW.EMPLOYEES program rather than using the Pick EDITor. This ensures the integrity of the data on the files, and enables adequate audit trails and other security procedures to be implemented. There are, however, many items for which it is not appropriate to use such applications software. These include:

- o Basic programs,
- o Procs, menus, and macros,
- o Attribute definitions, and phrases,
- o Test data items,

and such items will generally be created and maintained by the standard system software:

- o The Update Processor or the EDITor to create, change and delete the items,
- o Access and other TCL verbs to inspect and produce reports on the items.

Let's look at some of the ways in which you can handle these non-data items. In general, each item is handled individually, but many verbs will process a *sequence of items*, one after another. In such cases, the list of item-ids may be specified on the command which invokes the action, or a select-list may be produced immediately prior to invoking the action and the item-ids in that select-list will be processed.

4.8.1 Creating / changing items

Non-data items - such as macros, menus, Basic programs and Procs - will normally be created and changed by means of the Update Processor or the Editor, as we shall see later, using commands such as:

```
UPDATE file.name item.list
EDIT file.name item.list
```

New items may also be constructed by COPYing existing items and cannibalising these to produce new macros, menus, programs and Procs.

4.8.2 EXCHANGE: exchanging items

The EXCHANGE command:

```
EXCHANGE file.name aaa bbb
```

can be used to interchange two items on a file, so that the original contents of item *aaa* are now saved with item-id *bbb*, and the original contents of item *bbb* are now saved with item-id *aaa*.

4.8.3 Listing items

Reports on the contents of any Advanced Pick file can be produced by Access sentences of the form:

```
LIST file.name {select.criteria} {item.list}
SORT file.name {select.criteria} {item.list}
```

and so on, as we discuss in Chapter 12 where we look at the Access language. Some examples of Access sentences might be:

```
LIST MBPROGS
SORT MBPROGS = "SET.]"
LIST MBPROGS WITH A0 = "SET.]"
SORT MBPROGS = "[CALC]" "[ACCOUNT]"
LIST MBPROGS 'CALC001' 'CALC002' 'CALC003'
LIST MD = "[FLOPPY]" A1
```

Such sentences will produce a listing of the item-ids of the items on the file, as shown in Figure 4.25, and the report may include particular attributes.

The contents of the individual items can be displayed or printed by commands of the form:

```
CT file.name item.list
CP file.name item.list
```

using the CT (copy to the terminal) or CP (copy to the printer) verbs, or using Access sentences such as:

```
LIST-ITEM file.name {select.criteria} {item.list}
SORT-ITEM file.name {select.criteria} {item.list}
```

For example:

```
CT TESTDATA 1000 1100A 2300
CT MBCODE
LIST-ITEM MBPROCS
SORT-ITEM MBPROGS WITH A1 = "[SUBROUTINE]"
```

as illustrated in Figure 4.25.

In all these commands, the reports can be sent to the printer by adding the (P option:

```
LIST MBPROGS (P
SORT MBPROGS = "SET.]" (P
LIST MBPROGS = "[FLOPPY]" A1 (P
CT TESTDATA 1000 1100A 2300 (P
LIST-ITEM MBPROCS (P
SORT-ITEM MBPROGS WITH A1 = "[SUBROUTINE]" (P
```

```
:SORT MBPROGS = "SET.]"
```

```
MBPROGS...
```

```
SET
```

```
SET.ATTRIBUTE
```

```
SET.BOX.CHARS
```

```
SET.CURRENT.CHEQUE.BOOK
```

```
SET.DC.POINTERS
```

```
SET.LCD
```

```
SET.PASTES
```

```
SET.PRICES
```

```
SET.PRINTER
```

```
:LIST MBPROGS = "[FLOPPY]" A1
```

```
MBPROGS... A1.....
```

```
GFLOPPY1 SUB GFLOPPY1(DRIVE,DENSITY)
```

```
FLOPPY1 SUB FLOPPY1(DRIVE,DENSITY)
```

```
:CT MBPROGS CRT.DOT.LINE
```

```
CRT.DOT.LINE
```

```
001 SUB CRT.DOT.LINE(PART1,PART2)
```

```
002 DOTS=STR(' ',SYSTEM(2)-LEN(PART1)-LEN(PART2)-2)
```

```
003 CRT (PART1:' ':DOTS:' ':PART2)[1,SYSTEM(2)]
```

```
004 RETURN
```

Figure 4.25: Listing files and items

4.8.4 Deleting items

Items can be deleted from a file by means of the Update Processor, using the:

```
UPDATE MBPROGS TESTPROG
^XO
```

sequence, or by means of the Editor, using the:

```
EDIT MBPROGS TESTPROG
FD
```

sequence, or by means of the DELETE command:

```
DELETE MBPROGS TESTPROG
```

You may select and delete several items, as in the sequence:

```
SELECT MBPROGS = "TEST]"
DELETE MBPROGS
```

which would delete all programs with names beginning TEST.

In all cases, the item(s) will be deleted from the file and their contents lost. The TCL command:

```
RECOVER-FD
```

can be used to recover a **single item** which you have deleted by means of the Editor or the Update Processor (but not the DELETE command). This must be the next command you give and the process will ask you to confirm:

```
Recover "fffff iiii" (y/n)?
```

The item will then be reinstated as it was immediately prior to the EDIT or UPDATE command which deleted it. If you have deleted several items during one Editor or Update Processor session, only the last one will be recoverable.

As we have seen, all the items can be removed from a file by means of the CLEAR-FILE command.

4.8.5 COPY: copying items

The COPY verb allows you to copy items from one file to another and to display or print the contents of items.

The general form of the COPY command is:

```
COPY file.name1 {item.list1} {(options)}
```

where *file.name1* is the name of the file from which the items are to be taken, and *item.list1* is a list of item-ids of the items which are to be copied. If *file.name1* is omitted, then the process will abandon; if *item.list1* is omitted (or if an asterisk is used), then all items on *file.name1* will be copied; if *item.list1* is omitted and a select-list is available, the selected item-ids will be used.

Figure 4.26 shows some of the options which are available with the COPY verb. P and T are of special interest and print/display the contents of the items without performing any physical copy. If no P or T option is

specified, then it is assumed that the items are to be copied to a file. Some examples of the COPY command are:

```
COPY FILE1
```

```
COPY FILE1 *
```

Either of these forms may be used to copy all the items from FILE1 to another file. We'll see how to specify the destination file later.

- n** copy only the first *n* items.
- D** delete the source items after a successful copy. Not used with the T or P options.
- F** skip to a new page for each item. Used with the T or P options.
- I** suppress the display of the item-id.
- N** This option has two distinct actions: when used with the T option, it will suppress the pause at the end-of-page; when copying items to another file with the O option, it will only copy items which *already exist on the target file*.
- O** overwrite an existing item on the target file with the same item-id. Not used with the T or P options.
- P** print the items on the printer.
- S** suppress error message output when copying to a file, and suppress display of line-numbers when used with T or P options.
- T** display the items on the terminal.
- U** create a new, unique item-id - consisting of the original item-id followed by a letter - if the item-id already exists in the destination file. Not used with the T or P options.

Figure 4.26: COPY options

```
COPY FILE1 (T
```

```
COPY FILE1 * (T
```

either of these will display all the items of FILE1 on the terminal.

```
COPY FILE1 (100
```

to copy the first 100 items from FILE1 to another file.

`COPY FILE1 ITEM1 ITEM2 ITEM3 ITEM4`

to copy the four specified items from FILE1 to another file.

`COPY FILE1 ITEM1 ITEM2 ITEM3 ITEM4 (T`

to display the contents of the four specified items of FILE1 on the terminal.

`COPY FILE1 ITEM1 ITEM2 ITEM3 ITEM4 (P`

to print the contents of the four specified items of FILE1 on the printer.

`COPY FILE1 ITEM1 ITEM2 ITEM3 ITEM4 (OD`

to copy the four specified items from FILE1 to another file, overwrite any duplicate items on the destination file, and delete the source items when they have been copied.

A select list may feed the COPY statement:

`SSELECT FILE1 WITH BALANCE = "0"`

`COPY FILE (D`

to select those items which satisfy the selection criterion (`WITH BALANCE = "0"`) and copy these to another file, deleting the original items from FILE1 when they have been successfully copied.

Unless the T and P options are used, the process assumes that the items are to be copied to a file, and asks for the name of the file and the item-ids of the new items on that file:

TO:

You must then type in the **destination** of the items. There are several possible responses:

Response 1: <Return>

`COPY FILE1 A B C`

TO:

<== hit <Return> here

If you just hit <Return>, the items will be displayed on the terminal, exactly like the effect of the T option.

Response 2: enter the name of the file to which the items are to be copied:

`COPY FILE1 A B C`

TO: (FILE2

will copy item A of FILE1 to create item A on FILE2, item B of FILE1 to create item B on FILE2, and so on. **Note that the open parenthesis indicates that a file name follows.**

If the O option had been specified on the COPY command, then any

existing items A or B or C on the file.name2 would be overwritten. If the O option had not been specified, and items A, B or C already existed on FILE2, a message would be displayed and the duplicate item(s) ignored. If the D option had been specified, items A, B and C would be deleted after they have been copied successfully.

Response 3: enter the name of the file and the item-ids to which the items are to be copied:

COPY FILE1 A B C

TO: (FILE2 AAA BBB CCC

will copy item A of FILE1 to create item AAA on FILE2, item B of FILE1 to create item BBB on FILE2, and so on.

Note that the open parenthesis indicates that a file name follows.

Response 4: enter a list of item-ids with which the items are to be saved on the same file:

COPY FILE1 A B C

TO: AAA BBB CCC

will copy item A of FILE1 to create item AAA on FILE1, item B of FILE1 to create item BBB on FILE1, and so on. If the D option had been specified:

COPY FILE1 A B C (D

TO: AAA BBB CCC

will effectively rename the items, changing the item-id of A to AAA, B to BBB, and C to CCC.

In all situations, there is an implied one-to-one correspondence between the item-ids in the source item-list and those in the target item-list.

Take care! Note that only the open parenthesis distinguishes the Response 3 form:

(FILE2 ITEMSA ITEMSB ITEMC

from the Response 4 form:

ITEMA ITEMSB ITEMC ITEM D

If the open parenthesis is omitted, the response will be assumed to be of the Response 4 type and the items will be copied back on to the file.name1. **This is a common source of error.**

An extension to the general principles described here uses the **primary** and **secondary select-lists**. If a primary select-list is available, this can be used instead of the *item.list1*, and if a secondary select-list is available, this can be used instead of the *item.list2*. Look at this sequence:

SSELECT SFILE = "ORIG]"

```
SSELECT TFILE = "OLD]" (S
COPY SFILE (OD
TO: (TFILE
```

which will select all the items on SFILE whose item-ids start with ORIG (selected as the primary select-list) and copy them to TFILE where they will overwrite all the items whose item-ids start with OLD (selected as the secondary select-list). This assumes that SFILE holds items such as ORIG01, ORIG02, ORIG03 and so on, and TFILE holds items OLD01, OLD02, OLD03 and so on, and that there are equal numbers of items in the two sets.

4.8.6 COMPARE: comparing files

The COMPARE verb will let you compare the contents of two sets of items, attribute by attribute. The items may be on the same file or on two different files. The format of the command is:

```
COMPARE file.name1 {item.list1} {(options)}
```

specifying the name of the file and the list of items to be compared. If a select-list is available, then you can omit the item-list, and if you want to compare *all* the items on the file, then you can omit the item-list or use an asterisk in place of the item-list.

The output display comprises two columns, one for each file, showing the results for each item. Alongside each attribute, the code I indicates an extra attribute, and D indicates a missing attribute.

The options control the manner in which the comparison is carried out and the format of the output, and include:

- A compare the whole of each attribute. Without this option, only the first three words of each attribute are compared.
- C combine the output into a single column format.
- S suppress the output of the item-ids.
- Z output only those attributes which differ..

Like the COPY verb, COMPARE requires the name of a second file and a second list of items. Here are some examples:

```
COMPARE FILE1 A B C
WITH: (FILE2
```

will compare item A on FILE1 with item A on FILE2, item B on FILE1 with item B on FILE2, and so on. **Note that the open parenthesis indicates that a file name follows.**

```
COMPARE FILE1 A B C
WITH: (FILE2 AAA BBB CCC
```

will compare item A on FILE1 with item AAA on FILE2, item BBB on FILE1 with item B on FILE2, and so on. **Note that the open parenthesis indicates that a file name follows.**

```
COMPARE FILE1 A B C
```

```
WITH: AAA BBB CCC
```

will compare item A on FILE1 with item AAA on FILE1, item B on FILE1 with item BBB on FILE1, and so on.

There should be a one-to-one correspondence between the item-ids in the original item-list and those in the second item-list.

Like the COPY verb, COMPARE is able to exploit the primary and secondary select-lists. A primary select-list may be used instead of *item.list1*, and a secondary select-list instead of *item.list2*. For example, the sequence:

```
SSELECT SFILE = "OLD]"
SSELECT TFILE = "NEW]" (S
COMPARE SFILE (AZS
TO: (TFILE
```

will select all the items on SFILE whose item-ids start with OLD and compare their contents with all those items on TFILE whose item-ids start with NEW.

4.8.7 SEARCH-FILE / SEARCH-SYSTEM: finding strings

The SEARCH-FILE command (or the FIND equivalent command) will scan all the items on a file and produce a saved-list of the item-ids of those items which contain any of a number of strings. The process is invoked by either the SEARCH-FILE or the FIND command, and the process will ask a number of questions:

- **Source file to search:** enter the name of the file to be searched.
- **String to search for:** enter, one at a time, the strings which you are looking for, entering null when there are no further strings. The process will accept up to 15 strings.
- **Enter list item name:** enter the name by which the list of item-ids is to be saved on the POINTER-FILE.

There is also the SEARCH-SYSTEM command (or the equivalent FIND2 command) which will search through all the files on a specific account or the entire system, looking for specific strings.

4.9 Summary

Advanced Pick data is organised as items held on files. Each item is uniquely identified by its item-id, and consists of attributes; each attribute consists of values; each value consists of subvalues.

A physical file consists of one or more groups of frames of virtual memory. As items are written to the file, the hashing algorithm uses the item-id to determine which group is to hold the item. Within each group, the items are placed sequentially, one after the others. As a group overflows, another unused frame is linked on to the end of the group.

As each item is written to disk, it is prefixed by an 8-byte control field which records the nature of the item, its length and whether it has been changed/deleted since the last file-save.

Each logical file may consist of a DICT section and a data section, a DICT section only, or a DICT section and several data sections. The files on the system are organised into accounts.

When an account is created, a D-pointer is set up on the MDS file. This item records the name of the account, the location and the modulo of the MD of the account, and enables Advanced Pick to locate the account and its MD. When a file is created, a D-pointer is set up on the MD of the account. This item records the name, the location and the modulo of the file, and enables Advanced Pick to locate the file. For each data section, the DICT section holds a D-pointer which enables AP to locate that data section. Details of all files on the system are recorded on the FILE-OF-FILES file.

There are a large number of standard files owned by the DM account and these are made available to other accounts through pathnames or via a set of file-synonyms.

Files and items can be created / inspected / listed / copied / maintained / deleted by TCL commands – including Access sentences and the Update Processor – and by Basic programs.

5

File design

The tasks which face the Systems Analyst when designing an application for an Advanced Pick system are no different from those encountered when working in any other environment. The gathering, assembly and analysis of information, and the development of data models and process models are virtually identical on all computer systems. In this Chapter, we are concerned with the special points of interest which are relevant when the analyst is transforming the **logical entities** and **datastores** of the design into the physical files which will be used by the system.

Before we look at the physical design, we should mention that analysts who follow the principles of **data normalisation** propounded by **Codd** and others, will find that it is feasible to produce a **relational database** within the Advanced Pick environment. They will, however, be surprised to find that **repeating groups** are quite respectable and used widely – even to the point of allowing repeating groups within repeating groups! These repeating groups – or multivalued attributes – are a convenient means of holding the sort of data which crops up repeatedly in commercial applications, data such as the order lines of an invoice within an INVOICE item, the numbers of all the orders placed by a client within the CLIENT item, and so on.

We have already met some of the terms used in Advanced Pick: **account; file; item; item-id; attribute; value; subvalue**. The Systems Analyst will be concerned with the way in which these concepts are applied to holding the data within the items of the various files within the application system. There are some general points which the analyst should bear in mind:

- There may be any number of accounts on the system.
- There may be any number of files on each account.
- There will be one dictionary for each file.
- Less commonly, there may be several data sections to each file.
- There may be any number of items on each file.
- An item may be of any length. It is good practice to keep the items as small as possible.
- Each item may consist of any number of attributes.
- Each attribute may consist of any number of values.
- Less commonly, each value may consist of any number of subvalues.
- When there are a large number of items on a file, the correct choice of modulo can improve the efficiency in handling the file.

5.1 File names

There are fewer constraints to file names on Advanced Pick than on other computer systems. Some of the points to observe when choosing names for your files include:

- A file name may be up to 51 characters in length.
- Keep the names short but meaningful and memorable.
- Many analysts adopt a name structure such as:

```
INVOICES.1994
VOUCHERS.1994.CLEARED
VOUCHERS.1994.CURRENT
SALES-OCT
```

which is clear and explicit but can lead to typing errors if these are carried over to the live system. Non-technical users – or indeed anyone who is typing Access sentences to interrogate the file – may find these names difficult to remember. In such cases ...

- File-synonyms may be used to give more user-friendly names to your files.
- The file name must not include spaces, but any other characters may be used, however ...
- Avoid the characters:

```
" ' \ [ ] * ( ) ,
```

in file names as these might be ambiguous in certain contexts. The AP field separators – attribute-mark, value-mark, subvalue-mark and segment-mark – cannot be used in file names..

5.2 Modulo and file efficiency

The modulo of a file has a profound effect on the speed and efficiency of the processes which handle the file. In almost every situation, it is the actual physical read – getting the frame from disk – which determines the efficiency of the process: *the fewer the reads, the faster the process*.

There are two main methods of accessing the items on a file: sequential access and direct access.

If you are using Access sentences such as:

```
LIST MD SORT STOCK WITH MATERIAL "PINE"
```

or the Basic SELECT statement, then these activities process *all* the items on the file **sequentially**. In order to be efficient, you need to be able to process as many items as possible in as few reads as possible. This

implies that it is important to have each frame filled so that each read gives the best possible value in terms of number of items read in, as illustrated in Figure 5.1.

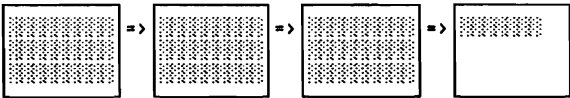


Figure 5.1: Sequential processing

In contrast, if you are using Access sentences such as:

```
LIST STOCK '1000' '2000' '3000'
```

a command such as:

```
UPDATE STOCK '1000'
```

a sequence such as:

```
GET-LIST REDS
SORT STOCK
```

or the Basic READ statement, then you will process *individual* items on the file **directly**, one at a time, using the item-id to find each individual item, and you need to be able to find each item as quickly as possible with as few unnecessary reads as possible. This implies that it is important that the processor should find the required item immediately it reads in the first frame of the group – that is, the groups should not overflow – as this would require a further read to find the item. In this context, model (a) is unsuitable and a good design might be that shown in Figure 5.2.



Figure 5.2

The decision is then based upon which of these – seemingly conflicting – modes of using the file is the more important to you. In general, it is likely that a compromise will be made, satisfying both demands as far as possible. For efficient file organisation, therefore, the requirements are:

- Have each frame filled with items; this gets the greatest benefit from each disk-read and makes best use of the disk space.
- Have no empty groups in the file; this avoids useless reads when processing sequentially.
- Have no overflow in any group; this makes direct access more efficient.

If the items are larger than 847 bytes (or 1694 on some systems), they will be held indirectly. As a consequence of this:

- Very large items are only represented by the pointer, and behave like

very small items.

- o Each large item will require one read through the primary file space (to find the pointer), and then one or more further reads (to retrieve the true data).

and, in some circumstances, the best solution might be to design the items so that they are 847 bytes or less in size to avoid the second set of read actions.

These factors should be borne in mind throughout the file design activity.

5.3 Choosing a modulo

The ideal for any file is that all groups should be filled but not overflowing. This suggests a possible algorithm for determining the best modulo for a file:

- 1) Determine the number of bytes in one frame of your system; let us call this FRAME.SIZE. FRAME.SIZE will be 1000 or 2000.
- 2) Use the results of your analysis to determine the average size of the items on the file; let us call this ITEM.SIZE.

When working out the item size, remember that a **direct** item comprises the 8-byte control field, the item-id and the data (and separators) of the item. This is padded with `_` characters so that the entire string is an even number of bytes long. You may use the formula:

$$\text{ITEM.SIZE} = 11 + i + d + n$$

where i is the length of the item-id, d is the length of the data in the item, and n is an integer (0 or 1) such that the total is a multiple of 2.

An **indirect** item comprises the 8-byte control field, the item-id and the 6-byte pointer (and separators). You may use the formula:

$$\text{ITEM.SIZE} = 17 + i + d + n$$

where i is the length of the item-id, d is the length of the data in the item, and n is an integer (0 or 1) such that the total is a multiple of 2.

- i) Use the results of your analysis to determine the (approximate) number of items on the file; let us call this ITEM.COUNT.
- j) Calculate the number of items which can be packed into one frame; let us call this PACKING:

$$\text{PACKING} = \text{FRAME.SIZE} / \text{ITEM.SIZE}$$

If PACKING is not an integer, then it should be rounded *up* to the nearest integer.

- 5) Calculate MOD, the number of groups in the file – this is the **modulo** of the file:

$$\text{MOD} = \text{ITEM.COUNT} / \text{PACKING}$$

You may care to adjust the calculated value of MOD, making it an odd number or a prime number, as discussed later.

You now have the modulo for the file. Two examples of the calculation are illustrated in Figure 5.3

ITEM.SIZE=100 bytes ITEM.COUNT=10000 items FRAME.SIZE=1000 bytes PACKING=FRAME.SIZE/ITEM.SIZE = 1000 / 100 = 10 MOD=ITEM.COUNT / PACKING = 10000 / 10 = 200 = 211 (nearest prime number)	ITEM.SIZE=700 bytes ITEM.COUNT=1000 items FRAME.SIZE=1000 bytes PACKING=FRAME.SIZE/ITEM.SIZE = 1000 / 700 = 1.428 = 2 (rounded up) MOD=ITEM.COUNT / PACKING = 1000 / 2 = 500 = 503 (nearest prime number)

Figure 5.3 Calculating the modulo

This modulo may then be used when creating a new file, or when setting the reallocation value, as described in Chapter 21. If the file already exists, the modulo can be used as a starting value for a HASH-TEST investigation into the distribution of the items through the file.

Many Pick users believe that the modulo must be a prime number; this is not true on Advanced Pick. You may have any modulo for a file. Indeed, if the item-ids are all numeric, sequential and of the same length, virtually *any* modulo will give even distribution of the items across the groups of the file.

```
:PRIME 173
173 is a prime number
The next highest prime number is = 179
The next lowest prime number is = 167

:PRIME 200
The next highest prime number is = 211
The next lowest prime number is = 199

:PRIME
PRIME . computes the next greater prime number

Enter a positive integer = 200
The next highest prime number is = 211
The next lowest prime number is = 199
```

Figure 5.4 PRIME utility

The Pick hashing algorithm produces patchy distribution of certain item-ids, with **bunching** (that is, having some groups over-populated and others empty) if you choose a modulo which is a factor of 10 (that is, 2 or 5 or 10), but that does not imply that you may not use such a modulo. To avoid such patchy distribution, the *rule of thumb* has been to choose an odd number, or better still, a prime number for the modulo. If you wish to follow this rule, then you would choose the nearest prime number which is equal to or greater than the value for MOD derived from the above algorithm. Advanced Pick offer the PRIME utility for determining prime numbers, as illustrated in Figure 5.4

5.4 Dictionary design

When you are using the Access enquiry language, the vocabulary of the sentences is made up of datanames located on the dictionary – the DICT section of the file. The following points relate to the contents and nature of the items held on the dictionaries:

- There may be as many definitions on your dictionary as you require, and ...
- There may be more than one definition for any particular attribute of the data items, but ...
- Only have those definitions which are necessary.
- Do not store large items – such as Procs – on the dictionary. Store these on a separate file.

Some system generation tools use the dictionary of the file to store information relating to the file, such as:

- The last invoice number used,
- Control totals,
- The raw specifications for the field or the screen layouts drawn up by the Systems Analyst,

and other pieces of control information relating to screen displays and reports. There is not much you can do about relocating such items.

- As with any file, a carefully selected value for the modulo of the DICT section of a file can improve its efficiency.
- The DICT section of a file is physically quite distinct from the data section and normally has a much smaller modulo than the data section.

When designing the datanames and the attribute definitions:

- Do not choose datanames which conflict with the standard Access keywords and throwaway modifiers. If you use datanames such as BY, TOTAL, WITH, THE and so on, they will be ambiguous when you use them in Access sentences.

- Do not create datanames which contain the characters:
" ' /
as these will be ambiguous when used in Access sentences.
- Choose datanames which are consistent, meaningful and easy to remember.
- The **temporary attribute items** – A1, A2, A3 and so on – display the raw data of their respective attributes left-justified in as wide a column as possible. They are of value if no suitable dataname exists for a field.
- Provide a set of general-purpose definitions (on the MD so that they can be used with any file on that account) with datanames such as:
 - A0.15.R to display attribute 0 right-justified in a 15-character wide column,
 - GAP5 to output five spaces, giving a wider space between the columns,
 - VERT to output a null field in a 200-character wide column, forcing non-columnar output for a report,
 - *LINES to output the number of lines/attributes in the item,and so on.
- In the attribute definitions themselves, attribute 3 need only contain a text-heading when this differs from the dataname; attribute 4 will normally be null; attributes 5, and 6 will always be null.

5.5 Item-id design

There are a few points to bear in mind when you are designing the item-ids for the items on your files.

- Undesirable bunching of the items within the groups of a file can be minimised if the item-ids are created such that they are numeric and sequential. For example, items with item-ids 100, 101, 102, and so on, will be distributed more evenly through the groups than will those with item-ids SMITH.A, SMITH.S, JONES, JENKINS and so on.

For the best distribution, the item-ids should be:

- Numeric,
- Sequential,
- No item-ids missing from the sequence,
- All of the same length (and padded with leading zeroes where necessary).
- You may hold **meaningful data** within your item-id. This has two advantages: firstly, the information which makes up the item-id need not be held within the item, and secondly, you can determine a

particular item-id more easily. For example, we might have the item-ids of a financial transaction file based upon the account and the date. This method exploits the uniqueness of the item-id to advantage.

- Item-ids may be up to 51 bytes in length. Longer item-ids are truncated at the 51st character.
- Only the left-most 32-bytes of the item-id are used in the hashing algorithm. If you have item-ids longer than this (and it is quite possible if you exploit the ability to use meaningful item-ids), then the distinguishing elements should be placed at the right-hand end of the item-id.
- Avoid the characters:

" ' \ [] ()

and spaces in item-ids as these may be ambiguous in certain contexts, and may make the items inaccessible to certain TCL commands and Access sentences. The AP field separators – attribute-mark, value-mark, subvalue-mark and segment-mark – cannot be used in item-ids.

5.6 Data design

The main criterion in the design points laid out here is that the items on the file should be as small as possible. In this way, you will have the benefit of improved file-access time as well as the reduced file-space. The points to note when deciding the format and sequence of the data attributes within the items include:

- There may be any number of attributes in your items.
- The attributes may be of any length.
- Try to keep your items as small as possible.

There is no limit to the length of an item. We have seen that, an item which is 847 bytes or less in length, is held on the file in the normal manner; for larger items, only a pointer is held on the file and the actual data is held separately in one or more dedicated frames of virtual memory. This makes the file faster when reading an individual item (since you have to search through less data to find the item you want), but slower when you are scanning the contents of the items (since you have to find the pointer and then read the frames which hold the item). As a general rule, try to keep the data within your items shorter than 847 bytes.

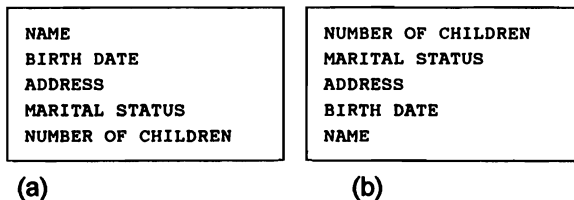


Figure 5.5 Sequence of data

- Attributes which are most likely to be empty, null or omitted, should be placed at the end of the item, arranging them in the order in which they are most likely to have actual values. Thus, the arrangement shown in Figure 5.5 (a) is preferable to that in Figure 5.5 (b) since many employees will have no children, but all employees will have a name. This practice should be adopted because Advanced Pick does not hold trailing null attributes nor does it hold the associated attribute marks.
- For numeric attributes, values of zero should be represented by null wherever possible. For most purposes, this will have the same significance as zero.
- Attributes which have a default – or most likely – value should represent this value by null. Thus, if an indicator represents MALE or FEMALE, then the most frequent (or likely) indicator, let us suppose this is MALE, should be represented by null, the other by 1 or F (but never by the word FEMALE).

This also applies to YES/NO indicators and flags, and may even be used in situations where an attribute may have one of a range of values. For example, let us suppose that attribute 7 of an item contains a code which may take any one of the values 0, 1, 2 or 3, and that the value 2 is the most common; then the actual values held on file should be 0, 1, null and 3, respectively, with the associated program, and Access conversions and correlatives, using logic such as:

```
IF 7 = "" THEN "2"
ELSE 7
IF N(CODE) = "" THEN
"2" ELSE N(CODE)
```

ADDRESSES	
100	GLEDHILL^34 HIGH ST
101	REECE^76 HIGH ST
102	MALLINDER^42 FLORAL AVE
103	JAGGER^9 FLORAL AVE

- If an attribute may have one of several known data values, it is more efficient to hold a *code* which cross-refers to a file containing the table of possible values, as illustrated by the street names in Figure 5.6.

(a) Explicit addresses

ADDRESSES	
101	GLEDHILL^34^H001
102	REECE^65^H001
103	MALLINDER^42^F910
104	JAGGER^9^F910

(b) Street codes

STREET.XREF	
H001	HIGH ST
F910	FLORAL AVE

(c)

This point would normally emerge during the data normalisation process, but the same principle could be used even more widely. A Systems Analyst designing the files for a local government

Figure 5.6: Holding codes

system might decide to hold the address explicitly, as in Figure 5.6 (a). But a better technique might be to represent the addresses by the street number and then a street code, as illustrated in (b), and using a separate cross-reference file which has items as in (c), and so on. This offers advantages of space and also of data consistency, in return for relatively small overheads on file-access times.

- Numeric attributes should be held without a decimal point. Thus, prices and amounts of money should be held in pence, and converted to pounds by a conversion code in Access and within Basic programs.

You may even wish to use **packed decimal** or **hexadecimal** values for certain numeric attributes. This offers advantages of security, protecting data such as salaries and telephone numbers from casual browsers.

- Dates and times will be held in the Pick **internal** format. Thus, the internal date 1234 takes up less space than any of the equivalent external formats 18 MAY 1971 or 18/05/71 or 18/5/71. Furthermore, the internal form is more convenient for sorting and for performing date calculations. With the conversion codes of Access and Basic, all of these external forms – and many more – can easily be reproduced from the internal form for output.
- Give some thought to the format of any textual data on your files. If in doubt, hold all your data in **UPPER-CASE**. We look at this again in Chapter 8 when we discuss the topic of case-sensitivity.
- Information held in the item-id should not be repeated within the body of the item. Within an attribute definition, the item-id is identified as attribute 0 and quite accessible for processing.
- If the items comprise data which is truly static, or rarely changed, together with data which is very volatile, it is more efficient to hold the two parts on separate files. The two parts would probably have the same item-id on their respective files.
- Avoid the characters:

" ' \ [] ^

in your data as these may be difficult to handle in Access selection criteria. For example, if a data attribute consisted of:

```
4' 8" wide [USA] standard
```

it would be difficult to select this by means of clauses such as:

```
LIST CODES WITH DESCR = "[4' 8"]"
LIST CODES WITH DESCR = "[[USA]]"
```

The AP field separators – attribute-mark, value-mark, subvalue-mark and segment-mark – should not be used in data fields.

- Once a file design has been decided and implemented, additional data attributes (such as may be required as a consequence of the users' changing requirements) may be added to the end of the item. Except where a Basic program uses an exactly sized dimensioned array in a MATREAD statement to read the data items, such additions should not affect any existing programs (or reports) which use the original attributes of the data items.

5.7 Security aspects

The security aspects of Advanced Pick allow you to control access to your files by means of the Ret-lock and Upd-lock **lock-codes**. If any of the D-pointers along the path to the data (from the SYSTEM file to the MD to the DICT section and on to the data section) contains a Ret-lock or Upd-lock lock-code, then any user attempting to access that file must have a matching value in the KEYS field of their USERS item. We discuss this in Chapter 23.

Advanced Pick does not allow you to apply lock-codes to a specific **attribute**, but only to an entire file section. Therefore, it may be necessary to reorganise your data, placing restricted data - salaries, addresses, telephone numbers - on a separate **locked** file if you wish to impose special security restrictions.

5.8 Summary

The general design procedures of modern systems analysis apply to Advanced Pick. But the file organisation and item structure of Advanced Pick together with its variable field length and variable item length offer great scope for an application to handle its data efficiently. Items may be extended as required during the life of the application without affecting existing systems in any way.

6

Using your Advanced Pick system

All your work on Advanced Pick will be done via the terminal: you will enter your information by typing on the keyboard, and the results will be displayed on the screen. There will also be one or more printers which you will use to produce printed reports. In this Chapter, I want to look at the ways in which a typical, non-technical user will work with these pieces of equipment.

So let's start by switching the terminal on.

6.1 Switching on

There are one or two elementary points to remember when you start to use your computer terminal:

- 1) Switch on the power point.
- 2) Switch on the terminal.
- 3) Adjust the brightness and contrast.
- 4) Press the <Return> or <Enter> key.

Advanced Pick will respond by displaying a message like that shown in Figure 6.1 (a). If this does not happen, check all the connections and then call your System Manager.

```
08:30:59  29 Jul 1994
Advanced Pick - Enter your Pick user-id:
```

(a) The system reacts

```
08:30:59  29 Jul 1994
Advanced Pick - Enter your Pick user-id: aaaaa
password: bbbbb
master dictionary: xxxxx
password: yyyy
```

(b) A typical logon sequence

Figure 6.1: Logging on to Advanced Pick

6.2 Logging on

Before you can carry out any work on the Advanced Pick system, you must **log on**. In order to log on to Advanced Pick, you will need several pieces of information:

- 1) Your **user-id**, and possibly ...
- 2) Your **user-password**.
- 3) The name of the **account** or the **master dictionary** which you will use, and possibly ..
- 4) The **password** for the account.

So a typical **logon sequence** might look like that in Figure 6.1 (b). In general, it does not matter whether you type your user-id, the account name or other information in UPPER-CASE or lower-case; Advanced Pick normally accepts either.

Your user-id is your own personal identifier and Advanced Pick uses this to identify your work, and also to keep a record of what you were doing when you last logged on to the system. The account which you log on to will determine which files and facilities are available to you. It may be that there are no passwords imposed on the user-ids and the accounts on your particular system. It is up to your System Manager to decide.

In some cases, your system may have been organised differently; it may be that – when you enter your user-id – Advanced Pick automatically logs you to the correct account, or you may asked for further identification, or you may be presented with a menu asking you what you want to do.

In all cases, your System Manager or your supervisor will provide you with the information you'll need.

Your user-id, the account names and the passwords are an important part of the security of the system, so it is good practice *not* to write such information down where anyone can see it.

We shall discuss user-ids and accounts in more detail later.

6.3 When you have logged on

When you log on to Advanced Pick, any one of several things may happen:

-) The computer may simply display the:

:

symbol and wait for you to issue a command. When the system is in this state, it is said to be at **TCL**. This means that Advanced Pick is waiting for you to type in an instruction in the **Terminal Control Language, TCL**.

We shall discuss TCL in more detail later.

- 2) There may be a request for some further identification. This may be a part of the security protection on your particular system.
- 3) You may be thrown straight into the business system for your organisation.

Exactly which of these things happens depends upon your particular system and the way in which your System Manager has set up the system.

6.3.1 TCL

TCL, the terminal control language, is the most fundamental means of using the system's facilities. The TCL language is made up a number of commands which are typed in, one at a time, whenever Advanced Pick displays the:

:

prompt. Normally, only technical users, such as programmers and analysts, will work at TCL. Most other users interact with the computer through an applications program which has been written for the specific work which they wish to perform.

Occasionally, however, non-technical users may be required to use TCL to make enquiries or produce reports about the data on their files. This is done by means of a special set of commands within the TCL language known as **Access**, the Pick enquiry language. The Access language uses **sentences** of the form:

```
LIST STOCK WITH MATERIAL "PINE" TOTAL PRICE
SORT EMPLOYEES BY NAME NAME DEPT AGE ADDRESS
```

Whenever you issue a TCL command it is carried out immediately – or it will be rejected if you have made a mistake.

We shall discuss all the aspects of Access in more detail later.

6.3.2 The stacker

The **TCL stacker** is a part of the TCL processor which allows you to recall the last TCL command which you entered and issue it again. You may use the stacker to repeat a particular command, or you may recall a command which you typed incorrectly and then correct it before issuing it again. This reduces the amount of typing which you have to do and makes it easier and faster to issue TCL commands and Access sentences.

The stacker holds the commands in a **stack**, hence the name. The most recent command is placed at the top of the stack. As you type another command, this is added to the top of the stack, pushing the others down. There is no limit to the number of commands which can be held in your

stack, and you can recall a command from anywhere in the stack.

If the stacker is in operation, you can use the keys shown in Figure 6.2 to recall, change and reissue your commands. Most useful are:

`^D`

to move down the stack starting with the most recent command, and:

`^F`

to move back up the stack. Keys such as these - and the other keys which we shall meet later - are typed by holding down the <Ctrl> key and pressing D (or F) at the same time. As you browse through the stack, each command will be displayed in turn; when the command you want is displayed, you simply press <Return> and that command will be executed, or you may use the other commands to change and correct the command before pressing <Return> and issuing the amended version.

<code>^D</code>	move down the stack
<code>^F</code>	move up the stack
<code><Return></code>	issue the displayed command
<code>^X</code>	ignore the displayed command
<code>xx^A</code>	find the most recent command containing the string xx
<code>^K</code>	move the cursor right one character
<code>^J</code>	move the cursor left one character
<code>^U</code>	move the cursor right one word
<code>^Y</code>	move the cursor left one word
<code>^G</code>	move to the end of the command
<code>^L</code>	delete the current character
<code>^O</code>	delete the current word
<code>^W</code>	insert a blank space
<code>^R</code>	start/stop inserting text

Figure 6.2: <Ctrl> keys : using the stacker

The contents of the stack are held indefinitely and Advanced Pick uses your user-id to save your stack for future use and to remember what you were doing when you last logged on.

6.3.4 In difficulties

Rarely, there may be occasions when you get into difficulty. The work you are doing may appear to run into trouble and stop with the system displaying something like:

```
I123
```

```
*
```

in which case your processing is said to have *gone into the Basic debugger*, or it might display something like:

```
I 123.45
```

```
!
```

or:

```
Icntrlnp:095
```

```
!
```

in which case your processing is said to have *gone into the system debugger*. We'll look at the debugging software in Chapter 18.

In another situation, the computer may appear to be doing nothing, or it may seem to be doing the same thing – like printing an error message – over and over again indefinitely. In this case, your supervisor or the System Manager may tell you to press the:

<Break>

key. The system will then respond with a prompt such as:

```
::
```

or with one of the two sequences shown above. In all situations, your supervisor or the System Manager will tell you what to do next. Typically, you may be told to type:

OFF to abandon your work and log off the system completely, or ...

END to terminate the work you were doing and go back to TCL or to the start of your application system.

You should always consult your supervisor or the System Manager before using either of these commands, otherwise you may lose some of your data or you may undo a lot of work which you or someone else has performed.

6.4 Logging off

When you have finished all your work and you no longer need to use the Advanced Pick system, you will **log off**. This instructs the computer to ignore you and your terminal until you (or someone else) logs on again.

If you are working within an application system, then there will probably be some formal way of finishing your processing and logging off. If you are using TCL, then you will log off by entering the:

OFF

command. You may then switch off your terminal, or you may wish to log on again later.

If you are working at TCL and want to log on to another account, you may do this directly by entering a command such as:

LOGTO WAGES

or, more simply:

TO WAGES

either of which will terminate the work on the current account and then log you on to the WAGES account. If there is a password on the WAGES account, you will be asked to enter this. Your user-id will be unchanged.

6.5 Switching off

If you have finished using your terminal – you may have finished work for the time being, you may be going to lunch, or you may be going home at the end of the day – then you will:

- 1) Log off, as we have just described,
- 2) Switch off the terminal, and finally ...
- 3) Switch off the power at the wall socket.

Your supervisor or the System Manager will tell you of any other security measures which apply to you.

6.6 Using your application system

The work which you do between logging on and logging off obviously depends upon your particular circumstances: yours may be sales organisation selling tables and chairs, you may work in a hospital handling patients and doctors, yours may be a public service company ... the range of Advanced Pick systems is limitless. For this reason, it is impossible to discuss your precise application in any detail, but in this and the following sections, I shall present some of the questions which you should ask of your supervisor so that you will be equipped to use your business application system.

- What is the name of the system?
- What is the system used for?
- Which terminal(s) will I use?
- Are there any restrictions on when and where I can use the system?
- How will I log on to the system?
- What is my user-id?
- Is there a password? Don't write down the password
- Which account (or accounts) will I use?
- Is there a password? Don't write down the password.

- Whom do I ask about any problems concerning the computer, the terminal, the printers?

For each individual task which I perform:

- What is the task called?
- What is the purpose of the task?
- Where does the data and/or the forms which feed the task come from?
- What is the data about?
- Whom do I ask about any problems concerning the task or the data?
- Where do the forms go when I have used them?
- Where do any reports go when I have produced them?
- How do I log off the system?
- Are there any other procedures to follow?

6.6.1 Using your application system

Many application systems are organised as some sort of **menu structure**, and the facilities within the menu structure are called up by selecting from a number of **choices** or **options** which are displayed at each menu.

Here are some questions which you should ask so that you will be equipped to use the menu structure of your business application system.

- How do I start up the menu system? Is the menu system called up automatically when I log on?
- How do I select an option from a menu?
- How do I go back up the menu tree?
- Can I jump from one branch of the structure to another, from one menu to another? If so, how?
- Can I jump straight to the top of the menu tree? If so, how?
- What help facilities are there?
- How can I ask for help?
- Can I log off from within the menu tree? If so, how?
- What can I do if I get lost within the menu structure?

6.6.2 The printers on your system

Here are some questions which you should ask so that you will be equipped to use the printers on your system.

- How many printers are there?

- Must I use a specific printer?
- Where are the printer(s) located?
- What commands must I issue (or what other action must I perform) when I want to use a printer?
- Are there any other points or procedures that I should know about using the printers?
- Will I use any special printer stationery for my work?
- If so, what should I know about this?

6.7 Summary

All work on Advanced Pick is done by typing commands and data at the keyboard and by reading messages and results displayed on the screen or printed on the printer.

In this Chapter, we looked at the mechanical aspects of using the system, and at the procedures for logging on and off, and for using a typical business application system.

7

The Advanced Pick keyboard

The terminal keyboard is used on Advanced Pick in much the same manner as on any other computer system with the following observations:

- The **function keys** and the **cursor control (arrow) keys** may be used within application systems but they are not used by Advanced Pick itself nor by the standard tools, such as the Update Processor or the TCL stacker.
- The **numeric keypad** may be used to enter numbers when the <Num Lock> key is set on.
- The special keys, such as <Page Up> and <Page Down> are not used by Advanced Pick.

Within Advanced Pick, the keyboard is given extended functionality beyond that encountered on R83 Pick systems, with great use being made of the <Ctrl> key combinations. The new actions apply whenever you are using:

- **The TCL stacker.** You will use the extended keyboard facilities to browse through the stack of commands issued earlier and to recall and to amend the commands.
- **The Update Processor** and all the system tools — such as CREATE-ACCOUNT — which use the Update Processor. You will use the extended keyboard to invoke the various actions, to create, change, maintain and/or delete the items and the data fields on your files.

This means that the functions of the TCL stacker and the Update Processor are invoked and controlled by typing <Ctrl> key sequences, and not by explicit commands (such as are used by the Pick Editor). Typically, these control sequences consist of the <Ctrl> key followed by one or more letters, and it is **not possible to use these Advanced Pick tools other than by means of these <Ctrl> keys.**

The <Ctrl> keys and keyboard sequences allow you to move through a data item and to perform a large number of editing and manipulating actions. For example, the key which is shown in this book as:

^K

or, to use the underlined notation adopted in some Advanced Pick literature:

k

is typed by holding down the <Ctrl> key and, at the same time, pressing the K key; this particular key will *move the cursor one place to the right*. Another example is the key sequence:

^XF

which is typed by holding down the <Ctrl> key and then pressing the X key and then releasing the <Ctrl> key and pressing the F key, or you may hold down the <Ctrl> key whilst both X and F are pressed. This particular sequence will instruct the Update Processor to *file the item* which is being edited.

Note that:

- You do not press the <Return> key or the <Enter> key when using these <Ctrl> keys and <Ctrl> key sequences.
- The letters may be entered as UPPER- or lower-case.

Here are some more examples:

^J

will move the cursor one place to the left.

^L

will delete a single character.

^W

will insert a single space.

^R

will toggle between insert and overwrite when you are entering text.

^XO

will delete the item.

^M

is equivalent to the <Return> key.

There are a great many such keys and key sequences. A table of the <Ctrl> keys is shown in the Appendix at the end of the book. At this stage, you should not worry if all these keys seems confusing ... take it from me,

with a little practice, they will all become second nature to you.

7.1 <Ctrl> S

It is worth reminding ourselves that, if you hit the:

`^S`

key, your terminal may apparently lock out, ignoring anything that you type in. This can be cancelled by typing the:

`^Q`

key. Advanced Pick does not use the `^S` key but it may be typed by accident when entering other keys and sequences such as `^FS..`

7.2 Redefining the user keyboard

The <Ctrl> keys and key sequences which we have discussed are the default settings used to control the Update Processor and other tools of Advanced Pick. In general, they are fixed and cannot be changed or reassigned to, say, the function keys.

Figure 7.1 shows a keyboard definition which might be used to assign the following actions to the <Cursor control> keys of the keyboard on port 0 for use with Advanced Pick:

- o <Up arrow> to move the cursor up one line (like the `^B` key).
- o <Down arrow> to move the cursor down one line (like the `^N` key).
- o <Left arrow> to move the cursor one character to the left (like the `^J` key).
- o <Right arrow> to move the cursor one character to the right (like the `^K` key).
- o `^<Up arrow>` to move the cursor to the top of the item (like the `^T` key).
- o `^<Down arrow>` to move the cursor down one page (like the `^T` key).
- o `^<Left arrow>` to move the cursor one word to the left (like the `^Y` key).
- o `^<Right arrow>` to move the cursor one word to the right (like the `^U` key).

This item is based upon the standard BRITISH-101 keyboard definition with the highlighted changes. Attributes 33 and beyond are unchanged. This item may be created and saved on your KEYBOARDS file, and invoked by means of the:

command, when required. Similar items could be used for other languages: USA-101, USA-84, FRENCH-84, and so on.

File: KEYBOARDS / Item: MB-AP-BRITISH-101	
01	* BRITISH 101-key keyboard for use with AP
02	*
03	* Upper case definitions
04	*
05	FF1B21402324255E262A28295F2B08FF
06	51574552545955494F507B7D0DFF4153
07	444647484A4B4C3A227EFF7C5A584356
08	424E4D3C3E3FFF2AFF20FF5051525354
09	5556575859FFFF3702392D0A350B2B31 !! Changed
10	0E33302EFFFFFFFF5A5BFFFFFFFFFFFFFF !! Changed
11	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
12	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
13	*
14	* Lower case definitions
15	*
16	FF1B313233343536373839302D3D0809
17	71776572747975696F705B5D0DFF6173
18	646667686A6B6C3B2760FF5C7A786376
19	626E6D2C2E2FFF2AFF20FF4041424344
20	4546474849FFFFFFFF02FF2D0AFF0B2BFF !! Changed
21	0EFFFFFFFFFFFF4A4BFFFFFFFFFFFFFF !! Changed
22	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
23	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
24	*
25	* Control key definitions
26	*
27	FF1BFF00FFFFFFFF1EFFFFFFFF1FFF7FFF
28	11170512141915090F101B1D0AFF0113
29	040607080A0B0CFFFFFFFF1C1A180316
30	020E0DFFFFFFFFFFFF20FF2021222324
31	2526272829FFFFFFFF14FFFF19FF15FFFF !! Changed
32	10FFFFFFFFFFFF2A2BFFFFFFFFFFFFFF !! Changed
33	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
34	FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Figure 7.1: A keyboard definition item

Note that, since each key can only send a single character, it is not possible to define a keyboard which would send sequences such as:

^XF

and so on. It is possible to redefine the function keys in a similar manner by means of the SET-FUNC command.

Release 6.1 and later have a facility for redefining the <Ctrl> keys by means of the SET-IMAP command which allows you to set up a mapping between a set of keyboard sequences and the equivalent internal codes used by the system.

7.3 SET-KBRD

The keyboard is set up to the USA layout when you boot the system. The SET-KBRD command allows the keyboard of the terminal on line 0 to be redefined for other layouts and for foreign languages.

A standard file – KEYBOARDS – is provided and contains a number of keyboard definitions for specific IBM national keyboards: BRITISH-84 / BRITISH-101 / BRITISH-102 / FRENCH-84 / FRENCH-102 / GERMAN-84 / GERMAN-102 / USA-84 / USA-101, and possibly others.

The keyboard can be set to any of these layouts by issuing a command such as:

```
SET-KBRD KEYBOARDS BRITISH-101
```

The items which define the keys consists of four sets of specifications representing the characters sent by the key and <Shift>, the key only, the key and <Ctrl>, and the key and <Alt>. Within each set, the lines represent the various characters generated by the keys on the top row of the keyboard, the second row, the third row, and so on. Each line is made up of pairs of hexadecimal digits, one pair for each key of that row. If the pair FF is used for any key, then that key will have no effect when pressed. The function keys are set by the SET-FUNC verb described later.

New keyboard definitions can be created and amended by means of the utility which is invoked by commands of the form:

```
SET-KEYS item.name
```

This will ask you for:

- o the **scan code** for each key which is to be set/changed. You will find a chart of these scan codes in the reference literature for your terminal keyboard.
- o the **hexadecimal character** which is to be sent when that key is pressed by itself, with the <Shift> key, with the <Ctrl> key, and with the

<Alt> key.

It is a good plan to copy one of the existing specifications and use this as a basis when you are creating a new set of keyboard assignments.

7.4 SET-FUNC

The function keys can be used with the Pick system. Their use, however, is different from their application within DOS systems, each function key sending a predetermined stream of characters.

The SET-FUNC command allows the functions keys F1 to F12 of the terminal on line 0 to be loaded with special characters.

An item defining the characters for each function key is established on a file, and this used to load the keys by means of the command:

```
SET-FUNC file.name item.name
```

specifying the names of the file and the item where the function key definition is held.

The item which defines the function keys consists of 4 lines each made up of 12 pairs of hexadecimal digits, one pair for each function key. The four lines represent the various characters generated when the user presses the function key and <Shift>, the function key only, the function key and <Ctrl>, and the function key and <Alt>. A fifth line specifies a pair of hexadecimal digits representing a **lead-in character** which will be transmitted immediately before the appropriate character for each function key.

A standard file – FUNCKEYS – is supplied with the AP system and contains a number of function key definition items – DEFAULT, NULL and TEST, and possibly others.

The command:

```
SET-FUNC FUNCKEYS DEFAULT
```

will reset the function keys to their default definitions, as shown in Figure 7.2; the asterisk represents the lead-in character, hexadecimal 2. If you are writing Basic programs to handle the function keys, it is perhaps wise to arrange for your program to receive these default settings and react accordingly.

Key	Unshifted	<Shift>	<Ctrl>	<Alt>
F1	*@	*P	*space	*0
F2	*A	*Q	*!	*1
F3	*B	*R	*"	*2
F4	*C	*S	*#	*3
F5	*D	*T	*\$	*4
F6	*E	*U	*%	*5
F7	*F	*V	*&	*6
F8	*G	*W	*'	*7
F9	*H	*X	* (	*8
F10	*I	*Y	*)	*9
F11	*J	*Z	**	*;
F12	*K	*[	*+	*;

Figure 7.2: Default function keys

7.5 Summary

Many of the standard Advanced Pick tools use the special <Ctrl> keys of the terminal keyboard. In this Chapter, we looked at some of the <Ctrl> key actions.

The SET-KBRD and SET-FUNC verbs are available to set the keyboard and the function keys to send specific character strings.

8

TCL and TCL commands

TCL, the **terminal control language** is the fundamental means of controlling and using Advanced Pick. TCL is used by typing a **TCL command** whenever the system displays the TCL prompt:

:

Each TCL command is executed immediately, and when the process has completed, the system returns to the TCL prompt.

The range of TCL commands allows you to perform a vast number of activities, including:

- Create and maintain data files and their contents.
- Create and maintain accounts.
- Copy data from one file to another.
- Create, maintain, compile and execute Basic language programs.
- Execute standard programs and utility software.
- Create, maintain and invoke macros, menus and procedures written in the Proc language.
- Control and interrogate the characteristics and the usage made of the system.
- Control the spooler and the printers.
- Perform backup and restore operations.

There are several forms of TCL commands. Some require only the command word:

TIME

others require more information:

DUMP 54321

and others require a file-name:

UPDATE STOCK

or a file-name and a list of item-ids:

```
UPDATE STOCK 1000 2000 3029 3948
```

Access sentences form a special subset of the TCL commands:

```
SORT STOCK WITH PRICE < "100" BY PRICE TOTAL VALUE
```

and are discussed separately in Chapter 12.

The first word in any TCL command is the **verb**. Each verb is defined by an item held on the **master dictionary** – the MD. Each separate account has its own MD and, therefore, its own repertoire of commands, although about 95% of all commands are available to all users; however, certain privileged operations, such as CREATE-ACCOUNT and DELETE-ACCOUNT, are normally only available to the DM account.

Figure 8.1 shows the MD items which define some of the common verbs. Exactly which system tool is used to produce the required effect is different for the verbs: the OFF verb and the UPDATE verb invoke special coding held in the ABS frames; the POVf verb calls a macro which then invokes another system tool, OVERFLOW; the LISTFILES verb invokes a Basic program held on the DM,BP, file. The ways in which a particular verb achieves its effect is of no concern to the ordinary user, the main thing is to know what the verb does and how to use it.

OFF	POVF	UPDATE	LISTFILES
01 VH	N	VE	VR
02 D	OVERFLOW	A]14	3]9
03			F
04			DM,BP, LISTFILES

Figure 8.1: Some TCL verb items

These verb-items are held on the MD exactly like the STAFF records are held on the STAFF file. Whenever a user types in a TCL command, the system picks out the first word – the verb – and then uses this to search through the MD to find the definition for that verb, and then invokes the appropriate tool. If the you make a mistake and type in a verb which is not defined on your MD, then it will be rejected with a message such as:

```
The verb 'xxxx' is not defined
```

If a particular account is not to be allowed to use any facility – such as the Editor – then the EDIT verb can be removed from the MD of that account. The verbs can also be renamed and you may even add your own programs to the MD so that they look and behave like TCL verbs and commands.

<code>^X</code>	cancel the command
<code>^K</code>	move the cursor right one character
<code>^J</code>	move the cursor left one character
<code>^U</code>	move the cursor right one word
<code>^Y</code>	move the cursor left one word
<code>^G</code>	move to the end of the command
<code>^E</code>	clear to the end of the command
<code>^L</code>	delete a character
<code>^O</code>	delete a word
<code>^W</code>	insert a blank space
<code>^R</code>	start/stop inserting text

Figure 8.2: <Ctrl> keys : correcting TCL commands

8.1 General notes on TCL

There are a few general points about working at TCL. We shall discuss some of these in more detail later.

- If you hit <Return> at the TCL prompt (that is, you enter a null command), then the TCL processor will display the time and the date and then repeat the TCL prompt. This display can be enabled/disabled by the TIMEDATE-ON and TIMEDATE-OFF commands:
- Any TCL command may be entered in either UPPER- or lower-case. Thus the sentences:

```
sort stock
SORT STOCK
sort STOCK
SoRt StOcK
Sort Stock
```

and so on, are all equivalent.

- There is no limit to the length of a TCL command.
- You can use the <Backspace> key in the normal manner, together with the <Ctrl> keys shown in Figure 8.2 to correct your commands as you type them in.
- The on-line HELP facility provided by the EPick (Encyclopedia Pick) can called for help on specific topics.
- Certain TCL commands will handle a list of item-ids, and if this is omitted, then *all* the items on the file will be processed:

```

EDIT STOCK
COPY STOCK
CT STOCK

```

R83 Pick systems used an asterisk to specify *all* items. This can still be used on Advanced Pick:

```

EDIT STOCK *
COPY STOCK *
CT STOCK *

```

and, if you wish to submit all the items on a file to the Update Processor, you *must* use the asterisk:

```

UPDATE STOCK *

```

Without the asterisk, a new item will be created.

- The **TCL stacker** will store each TCL command which the user enters. These lists of commands are stored on a file called TCL-STACK with the user-id as the item-id. The stored commands may be recalled and/or corrected as required.
- All the TCL commands issued by the user may be recorded on the CAPTCL file for inspection. This is a security feature employed in some organisations.

8.2 TCL options

Many TCL commands allow you to include a sequence of **options** to tailor the general action of the command in some way. The options are always specified at the *end of the command* and are preceded by an open parenthesis:

```

WHO (P
ISTAT (S
LIST-ITEM STAFF 1000 2000 (P

```

If several options are specified, these may appear in any sequence and with or without separating commas:

```

BASIC MYPROGS TEST (ESP
BASIC MYPROGS TEST (PSE
BASIC MYPROGS TEST (P,S,E

```

Each command has its own options and these will be shown in the technical description of the command, although for those verbs which produce a report, the P option and the N option are generally available:

P to send the output to the printer,

N to suppress the pause at the end-of-page when displaying on the screen.

For those commands which enquire / enable / disable specific features, such as:

```
BREAK-KEY
CASE
LEGEND
SPELLER
STACK
TCL-HDR
TIMEDATE
TYPE-AHEAD
```

the general options are:

F to disable the feature,

N to enable the feature,

and the command with no options will display the current status. For example:

```
TIMEDATE (F
TIMEDATE (N
TIMEDATE
```

and in most cases there will be explicit forms to enable/disable the feature:

```
TIMEDATE-OFF
TIMEDATE-ON
```

8.3 Case-sensitivity

We have seen that any TCL command and any Access sentence may be entered in either UPPER- or lower-case. The CASE command will allow you to reset and/or display the sensitivity for your own terminal:

CASE
will display a message such as:

```
[1318] Case sensitive.
[1319] Case insensitive.
```

showing the current status on your port.

```
CASE (N
CASE-ON
```

either of these will set your input to be case-sensitive.

CASE (F

CASE-OFF

either of these will set your input to be case-insensitive.

```
:case-off
```

```
[1319] Case insensitive.
```

```
:sort cfile with a1 "cat" a1
```

```
cfile..... a1.....
```

```
REC1      cat
```

```
REC2      Cat
```

```
REC3      CaT
```

```
REC4      cAT
```

```
REC5      CAT
```

```
[405] 5 items sorted out of 5 items.
```

```
:sort cfile with a1 "CAT" a1
```

```
cfile..... a1.....
```

```
REC1      cat
```

```
REC2      Cat
```

```
REC3      CaT
```

```
REC4      cAT
```

```
REC5      CAT
```

```
[405] 5 items sorted out of 5 items.
```

```
:case-on
```

```
[1318] Case sensitive.
```

```
:sort cfile with a1 "cat" a1
```



```
cfile..... a1.....

REC1      cat

[405] 1 items sorted out of 5 items.

:sort cfile with a1 "CAT" a1

cfile..... a1.....

REC5      CAT

[405] 1 items sorted out of 5 items.
```

Figure 8.3: Data case sensitivity

Figure 8.3 illustrates the way in which the case-sensitivity and the manner in which the data is held on the file can affect the results of Access sentences. Note that all commands and keywords are case-insensitive.

It is possible (by using a DS-pointer instead of the simple D-pointer for the data-level identifier) to define a file such that the item-ids are to be case-sensitive; that is, so that there may be four distinct items with item-ids:

```
FRED
Fred
fred
fReD
```

Whilst this may seem a good thing, it can lead to certain complications. In the absence of any other guidelines or constraints, it is good practice to:

- Have all your files case-insensitive.
- Have all your data and item-ids case-insensitive.
- Hold all your textual data in UPPER-CASE.
- Always work with the TCL CASE sensitivity off.
- Do not compile Basic programs with the CASING ON statement.

Many organisations enforce conventions such as these as standard practice.

8.4 TIMEDATE

If you just hit <Return> instead of entering a TCL command, the system will display **the time and the date**:

```
17:49:59  22 APR 1995
:
```

This action can be enabled/disabled by the commands:

```
TIMEDATE-ON
TIMEDATE-OFF
```

or the equivalent:

```
TIMEDATE (N
TIMEDATE (F
```

commands, respectively. The simple command:

```
TIMEDATE
```

with no options will display the current status of the function, indicating whether it is on or off.

8.5 Stacker

The Advanced Pick **stacker** stores every TCL command which you enter, and allows you to recall a TCL command which you entered earlier – and modify it, if you wish – and issue it again. The commands are stored in a **stack**. The contents of the stack can be scanned, inspected and recalled by the standard Update Processor <Ctrl> keys; these are summarised in Figure 8.4. Those marked with † may be used to amend a command as you are typing it.

^D	move down the stack	
^F	move up the stack	
<Return>	issue the displayed command	.
^X	ignore the displayed command	.
xx^A	find the latest command containing the string xx	
^K	move the cursor right one character	.
^J	move the cursor left one character	.
^U	move the cursor right one word	.
^Y	move the cursor left one word	.
^G	move to the end of the command	.

^L	delete a character	.
^G	clear to the end of the command	.
^O	delete a word	.
^W	insert a blank space	.
^R	start/stop inserting text	.

Figure 8.4: <Ctrl> keys : using TCL and the stacker

These keys and key sequences are entered at TCL. Of particular value are:

- You can recall the last command issued by means of the ^D key.
- You can navigate up and down the stack by means of the ^D and ^F keys.
- You can search for a specific substring within the stack by means of the ^A key. A command such as:

xxxxx^A

will search for the most recent command which contains the string xxxxx, and:

^A

will repeat the last search, going down the stack.

- You can reissue a command by pressing <Return> when the required command is reached and displayed.
- You can amend the displayed command by using the <Ctrl> keys for cursor movement and editing.
- You can delete the displayed command by means of the ^E key. This is useful for reducing the size of your stack.
- You can abandon the displayed command – without executing – by means of the ^X key.

The stacks of the users' TCL commands are held on the TCL-STACK file which is held on the DM account. Each individual user's stack is held with the user-id as the item-id.

There is no limit to the number of commands which are held in the stack, so you must take care to clear it down (or delete the item) from time to time by means of commands such as:

UPDATE TCL-STACK user.id

UPDATE DM.TCL-STACK, user.id

either of these will allow you to change the stack for a particular user.

```
DELETE TCL-STACK user.id
DELETE DM,TCL-STACK, user.id
```

clear down the entire stack for a particular user, or:

```
CLEAR-FILE TCL-STACK
CLEAR-FILE DM,TCL-STACK,
```

clear down the stacks for all users.

Commands invoked through a macro which has an M identifier (in attribute 1) are added to the stack, those in a macro with an N identifier are not.

The stacker is normally active, but it may be switched off/on by means of the commands:

```
STACK-OFF
STACK-ON
```

8.6 Capturing TCL history

It is possible to record all the TCL commands – including null commands – which the users issue. This ability to scan the recorded commands and monitor the user's work can be valuable for security purposes or when training new users.

For each user, a TCL command history item is saved on the CAPTCL file. These items hold the commands with the most recent at the end, as illustrated in Figure 8.5, and are held with item-ids of the form:

u*[p

where u is the user-id and p is the port-number, and [represents the character 127.

dm*[0

```
01 create-file captcl 1 11
02 list stock description material price
03 ud stock description material price
04 list-item captcl
05 list stock description material price
06 list stock description material price age
07 list stock description material price
08 list-item captcl
```

Figure 8.5: CAPTCL item

The capturing action is enabled by the TCL commands:

```
CAPTURE-ON
```

or the equivalent CAPT (N command, and is disabled and the recorded command history deleted by the command:

```
CAPTURE-OFF
```

or the equivalent CAPT (O command. The recording is invisible to the user and is quite distinct from that of the TCL stacker.

8.7 Level pushing

When you are carrying out any processing – be it a long Access report, an update program, or whatever – your terminal will be locked out until the process has finished and control returns to you. This can be inconvenient.

When you are in the midst of a process, you may press the <Break> key or the <Esc> key to suspend the current process and then issue a TCL command. When you interrupt a process in this manner, the system stops execution and saves all the necessary information so that the process can be resumed exactly where it was interrupted, it then replaces the normal system prompt by:

```
::
```

to indicate the level at which you are processing and waits for you to enter a TCL command. At this point, you may issue any TCL command to initiate a second process. This second process may, in turn, be interrupted and a further operation invoked. When each process has completed at the lower level, the appropriate prompt will be displayed, and you may issue a further TCL command at this lower level. This facility is known as **level pushing** and is illustrated in Figure 8.6.

```
:xxxx
```

Invoke process at level 1

```
xxxxxxxxxxxxxxxxxxxxx
xxxxx<Break>
```

Push to level 2

```
::yyyy
```

Invoke process at level 2

```
yyyyyyyyyyyyyyyyyyy
yyyyy<Break>
```

Push to level 3

```
:::zzzz
```

Invoke second process at level 3

```
zzzzzzzzzzzzzzzzzz
zzzzzzzzzzzzzzzzzz
_____ends
```

```
:::<Return>
```

Return to level 2

```

      YYYYYYYYYY
    YYYYYYYYYYYYYY
    YYYYYYYYYYYYYY
      ends
::aaaa                                Invoke process at level 2

    aaaaaaaaaaaaaa
    aaaaaaaaaaaaaa
    aaaaaaaaaaaaaa
      ends
::<Return>                            Return to level 1
      xxxxxxxxxx
    xxxxxxxxxxxxxxxx
    xxxxxxxxxxxxxxxx
      ends
:

```

Figure 8.6: Level pushing

The following special TCL responses are available when a:

```
:::
```

prompt is displayed:

<Return> to return to the previous level and continue execution of the process there.

END to terminate the process at the *lower* level.

OFF to log off the system and terminate all processes.

DEBUG to enter the system debugger, or, if the execution of a Basic program has been interrupted, to enter the Basic debugger.

Either the <Break> key or the <Esc> key may be assigned to invoke this level pushing, and this will normally be set during the logon procedure for the user or for the account. The following TCL commands are available to control this:

BRK-LEVEL
assigns the <Break> for level pushing,

ESC-LEVEL
assigns the <Esc> for level pushing,

If the ESC-LEVEL is enabled, you can still generate the <Esc> data character by means of the sequence ^[

BRK-DEBUG
assigns the <Break> for invoking the debugger (the conventional role),

ESC-DATA

assigns the <Esc> for sending the <ESCAPE> character (the conventional role).

ESC-TOGGLE

enables/disables the action of the <Esc> key between level pushing and sending the <ESCAPE> character, and vice versa.

Ln	PCB	PIB	ABS	Stat	R1	& Return stack contents	
	FID	Stat	Base				
000	00215A	FF10	000012	6	ws.where1:01E	ws.who:236	

depth of nesting —┐

Figure 8.7: Level pushing and WHERE

The depth of the processing level for each user is shown in the STAT field of the modified WHERE report, as illustrated in Figure 8.7.

The processing may be interrupted and nested up to 16 levels, beyond this depth, any interruptions will be ignored.

8.8 LOGTO and level pushing

When you interrupt your processing, it is possible to issue a LOGTO (or TO) command and log to another account. The TCL prompt on the new account will be of the form:

```
:::
```

to remind you that you have pushed a level. You may carry out any work on that account and when you hit <Return> at the prompt, you will return to your original processing on your original account.

8.9 Macros

If you need to issue the same TCL command or Access sentence, or the same sequence of commands and/or sentences, over and over again, it is possible to save the sequence as a **macro**. The macro can then be recalled by name, as and when required. This saves typing effort and removes the risk of errors, and is particularly valuable when developing systems for non-technical users.

The format of a macro definition is illustrated in Figure 8.8. In (b), STOCK.MACRO.1 invokes a single Access sentence, and in (c), STOCK.MACRO.2 invokes several commands one after another. Macro items such as these are created on the MD.

A macro is invoked by typing the name of the macro as if it were a TCL command. Thus, when the macro in (c) is invoked by means of the TCL command:

```
STOCK.MACRO.2
```

the macro processor will display each of the TCL commands in turn and wait for you to hit <Return> to execute the command. Before you hit <Return>, you may use the standard cursor controls of the Update Processor to amend any part of the display to patch the TCL command before it is invoked. The ^X command will ignore the command.

If the first line of the macro begins with N (instead of M), then the commands will be executed without requiring any user intervention.

The general format of a macro definition is illustrated in Figure 8.8 (a).

00 macro name
01 M or N {comments}
02 TCL command 1]data 1]data 2] ...
03 TCL command 2]data 1]data 2] ...
04 TCL command 3]data 1]data 2] ...
:
n TCL command m]data 1]data 2]

(a) Macro definition

STOCK.MACRO.1
01 M
02 SORT STOCK BY PART.NO

(b)

STOCK.MACRO.2
01 M THE STOCK CONTROL ROUTINES FOR THE STORES
02 SORT STOCK BY DESCRIPTION LPTR
03 COPY STOCK * (O)](STOCK.BACKUP
04 RUN PROGS STOCK.MAINT
05 CLEAR-FILE DATA STOCK.BACKUP

(c) several commands

Figure 8.8: Macros

The first attribute of the macro definition starts with the identifier M or N followed by optional comments. If M is used, each command in the macro is added to the TCL stack and passed to the user for him/her to modify and/or issue the command. If N is used, the commands of the macro are executed without requiring such user-intervention and they are not added to the stack.

Each attribute from attribute 2 onwards holds a TCL command, or a set of values representing the TCL command which is to be executed, followed by an optional stack of input data which is to feed the TCL command. A macro may contain any number of TCL commands.

If a macro is invoked by a command such as:

```
STOCK.MACRO.1 WITH MATERIAL "PINE"
```

then the specified text will be **appended** to the end of the first command in the macro, generating the sentence: `

```
SORT STOCK BY DESCRIPTION LPTR WITH MATERIAL "PINE"
```

in this instance. Beyond appending text to the first command in the macro, it is not possible to tailor the contents of a macro to incorporate data supplied at execution time; if it is required to modify the commands in any other manner, then you should consider using Procs. We discuss Procs in Chapter 17.

If any of the TCL commands (or the modified commands) is invalid, then the macro will abandon.

All macro definitions are held on the MD and will be created by means of the Update Processor or the EDITor:

```
UPDATE MD macro.name  
EDIT MD macro.name
```

A simple macro can be created by means of the CREATE-MACRO command:

```
CREATE-MACRO macro.name
```

or:

```
CREATE-MACRO macro.name (N
```

which will establish *the last executed TCL command* (taken from the top of your TCL stack) and save it on the MD as a macro with the name *macro.name*. The N option on the CREATE-MACRO command will establish the macro with an N identifier in attribute 1, otherwise M will be used.

Macros are held on the MD. Commands such as:

```
LIST-MACROS MD
LIST-MACROS WAGES,MD,
```

will list the available macros on the MD of your account or some other account.

8.9.1 Commands for use with macros / menus

A number of TCL commands are provided specifically for use in macros and menus: COMMENT / DISPLAY / PROMPT. The COMMENT command is used to display output on the screen and to produce special effects and cursor-positioning:

```
COMMENT {element} {element} ... {element} {+}
```

where the *elements* are any of:

- (c,r) to move the cursor to column *c* of row *r*. This is similar to the @(*c*,*r*) function in Basic and the (*c*,*r*) element in the Proc T statement, as we discuss later.
- (-n) to produce one of a range of special display effects. This is similar to the @(-*n*) function in Basic and the (-*n*) element in the Proc T statement, as we discuss later.
- (X*n*) to output the character whose decimal value is *n*. This is similar to the In element in the Proc T statement.
- (xxxx) to output the string *xxxx*.
- + to hold the cursor at the end of the output string.

The DISPLAY command has the form:

```
DISPLAY message
```

and will display the message on the screen. TYPE may be used as an alternative to DISPLAY.

The PROMPT command has the form:

```
PROMPT {message}
```

and will display the message:

```
{message} -- Quit/Continue (q/c) ?
```

on the screen and accept a response of Q or C from the user. A response of Q will abort the macro/menu, a response of C will continue with the macro/menu.

These commands can be issued directly at TCL, but they have little meaning outside the macro/menu context.

8.10 Menus

The concept of **menus** allows you to set up a system of TCL commands which can be invoked by menu selection. A typical example is illustrated in Figure 8.9 (b).

When this menu definition is invoked by means of the command:

```
STOCK.MENU
```

the menu processor will:

- Display the menu title,
- Display the available options and the number which will invoke each option,
- Display an input prompt message,
- Accept the user's control code and react accordingly,

as illustrated in Figure 8.10. You may then enter the required option:

- A response such as 2 will select option 2 (the PRINT STOCK ITEMS option) and invoke the TCL command:

```
SORT STOCK BY PART.NO LPTR
```

- A response such as 2? will display the appropriate help message:

```
Produce a printed list of the stock items
```

- A response of <Return> will drop out of the menu, and pass back to TCL.
- Any other response will be regarded as a TCL command and executed.

When the appropriate command has been executed, you will be asked to:

```
Hit any key to return to the Menu
```

and the menu will be redisplayed and the action repeated.

The general format of a menu definition is illustrated in Figure 8.9. The first attribute of the menu definition contains the identifier:

```
ME
```

followed by optional comments on the same line. The second attribute contains the title to be displayed at the head of the menu screen. For each option to be offered on the menu, attributes 3 onwards will hold a set of three values, representing:

- 1) A **descriptive text** which is to be shown alongside that option on the menu.
- 2) An optional **help message** which is to be displayed if you seek further information about that option.
- 3) The **TCL command** which that option is to invoke. This may be any TCL command and even another menu name. Several TCL commands may be specified as further *multivalues*.

```
00 menu name
01 ME
02 menu title
03 descriptive text 1]Help message 1]TCL command 1
04 descriptive text 2]Help message 2]TCL command 2
05 descriptive text 3]Help message 3]TCL command 3
:
n descriptive text m]Help message m]TCL command m
```

(a) Menu definition

```
00 STOCK.MENU
01 ME The stock control menu for the stores
02 Stock Control Processing
03 Display stock items
    Display a list of the items on the STOCK file
    SORT STOCK BY PART.NO
04 Print stock items
    Print a list of the items on the STOCK file
    SORT STOCK BY PART.NO LPTR
05 Add new stock items
    Add new items to the STOCK file
    RUN STOCK.BP ADD.REC
06 Archive stock items
    Archive out-of-date items on the STOCK file
    PROMPT Mount 1.44MB diskette in drive A:
    SET-FLOPPY (AH
    T-REW
    T-DUMP STOCK WITH AGE > "366"
```

```

T-DET
  DISPLAY Remove diskette
07 Delete stock items
  Delete old items from the STOCK file
  RUN STOCK.BP DEL.REC

```

(b) STOCK.MENU

Figure 8.9: Menus

The menu processor will handle any number of options, and the screen display will be tailored to handle the various lengths of descriptive text, producing only a single list if the texts are long.

It is not possible to tailor the TCL commands and/or Access sentences which are generated by a menu.

Stock Control Processing

1) Display stock items	4) Archive stock items
2) Print stock items	5) Delete stock items
3) Add new stock items	

Enter number of choice, number? for Help, <Return> to exit menu, or verb:

Figure 8.10: STOCK.MENU screen

Menu definitions are held on the MD and may be created by means of the Update Processor or the EDITor. There are two associated commands:

LIST-MENU MD {item.list}
to display the contents of the specified menu(s) held on the MD, and:

LIST-MENUS MD
LIST-MENUS WAGES,MD,
to list all the menus on your MD or the MD of another account.

As this last command will reveal, there are a number of standard menus provided with the Advanced Pick system.

8.11 Important TCL commands

In most commercial systems, users are rarely exposed to TCL. Most systems are designed so that users are thrown directly into the application system when they logon. Non-technical users who are allowed to TCL will predominantly use the Access verbs:

LIST
SORT

and a few TCL commands which are important to all users:

UPDATE
RUN
OFF
HELP
OP = the Output Processor

There are other commands which are of particular value to technical users:

EDIT
COPY
BASIC
COMPILE
CATALOG

As we discuss later, on those implementations which run within the DOS or Unix environments, it is possible to use the form:

!DIR

to issue a command from Advanced Pick TCL directly to the host operating system.

The System Manager is likely to make most use of the range of TCL commands which are provided to set, reset, assign, enable and disable the various characteristics of the Advanced Pick system. We discuss many of these commands individually in other Chapters of this book. We discuss some miscellaneous topics here.

8.11.1 EPick

EPick, *the Encyclopaedia Pick*, is a help facility offered with Advanced Pick. EPick provides information on every TCL command, Basic statement, Access component and a wide range of general Pick topics, and is supplied in hard-copy form and as an on-line reference source called up by the TCL HELP facility using commands of the form:

HELP topic
or:

? *topic*

where *topic* is the subject on which help is required. For example, the TCL command:

```
HELP FRAME
```

would invoke EPick and display the EPick entry on **frames**. If you ask for help on a topic which is not included in EPick, then the next available alphabetic entry will be offered. Thus, if you asked for:

```
HELP GRO
```

the EPick entry for GROUP would be displayed.

Having found a particular entry in EPick, you may use the AP keyboard sequences, shown in Figure 8.11, to browse through the information which is displayed.

To switch from one EPick entry to another, you should:

- o Move the cursor to the **TOKEN** field.
- o Press ^E to clear the current token.
- o Type in the topic (or just the first few letters of the topic) on which help is required.
- o Press ^F to jump to that topic.

You can terminate the current HELP session at any time by typing any of the ^X sequences shown in Figure 8.11.

Instructions for loading EPick accompany the diskettes of files and software. Having loaded the software, you should log to the EPICK account and a menu will be displayed allowing you to display information about the HELP system itself, use the HELP system for a specific topic, or install the HELP system on other accounts.

```
^F      cruise forwards through EPick
^D      cruise backwards through EPick
^G^G   zoom to a related subject
^XE     abandon the item
^XX     abandon the HELP activity
^XX     abandon the HELP activity
```

and the special **hotkeys**:

<code>^X4</code>	display the general HELP details
<code>^X5</code>	print a copy of the displayed information

Figure 8.11: <Ctrl> keys : using EPick HELP

8.11.2 SLEEP

All TCL commands are executed immediately they are entered, tying up the terminal until the job is complete and the system is ready for further commands. There are two ways in which this may be changed: the **SLEEP** command and the **phantom scheduler**.

The **SLEEP** command has forms such as:

```
SLEEP 22:30
    to sleep until the specified time, and
```

```
SLEEP 60
    to sleep for 60 seconds.
```

Whilst the user's process is sleeping, any information typed at the keyboard will be held in the type-ahead buffer until it can be accepted by the waking processes. For example, if you were to type:

```
SLEEP 22:30<Return>SORT STOCK (P<Return>OFF<Return>
```

your port would sleep until 10:30 p.m. and then wake up to print the **STOCK** report and finally, log off.

8.12 Phantom processes

The **Z** command invokes a **phantom process** and then allows you to continue with other work whilst the phantom process is carried out in the background under the control of the **phantom scheduler**. The general format of the command is:

```
Z{H}{S} {tcl.command}
```

for example:

```
Z LIST STOCK LPTR
ZH RUN WAGES.BP CALC.ALL
ZHS CALCULATE.YEAR.END
```

If **H** is specified (in either the **ZH** or **ZHS** forms), then all the terminal output will be sent to a spooler hold file; if the **S** is specified (**ZS** / **ZHS**), any error messages which may be generated by the phantom scheduler will be suppressed; if **S** is not specified (**Z** / **ZH**), all messages will be

displayed on the terminal from which the command was issued (this can be irritating).

If the *tcl.command* is omitted, you will be asked for:

- 1) The user-id (and the password) which is to be used. The default is the user-id which you are using at the moment.
- 2) The name of the account (and the password) which is to be used. The default is the account which you are using at the moment.
- 3) The TCL command which is to be invoked,
- 4) The number of the line on which the process is to run. The default is any available phantom line.
- 5) Any input data to be submitted to the command. A blank data line is represented by ^N, and the end of the input data is indicated by a null line.

The phantom process will then be initiated by the phantom scheduler and your terminal will be freed for normal use as the phantom job proceeds.

The JOBS file is used to record details of jobs handled by the phantom scheduler. The TCL commands STARTSCHED and STOPSCHEd are used to enable/disable the phantom scheduler.

A number of phantom ports are provided for use by the phantom scheduler; the number depends upon your particular configuration. These do not require the presence of a physical port.

8.13 GraPICK: the Pick GUI

GraPICK is a GUI – a *graphical user interface* – which uses **Microsoft Windows** software to present many Pick facilities in a visual manner which will be familiar to anyone who has worked in a Windows environment.

The features of GraPICK include:

- Mouse support,
- Update Processor front-end,
- A screen editor,
- Scroll bars,
- Text windows,
- Controlling and dependent windows,
- File zooming,

- Basic subroutines to incorporate the GraPICK features in your own systems.
- Graphical Access interface,
- PickLAN network support,

and many more. There will be more details of GraPICK in future editions of this book.

8.14 Summary

The Terminal Control Language is the low-level command-line interface for Advanced Pick. TCL commands are typed in and executed one at a time at the TCL colon prompt.

In most application systems, users will not be exposed to TCL but will invoke their processing via menus and other front-end processors.

The GraPICK software provides a graphical user interface for many of the standard AP tools and applications.

9

The Editor

The Editor is a fundamental tool for creating, changing and deleting any item held on any file. It was widely used on R83 Pick where it was the only hands-on tool for creating and editing items, but on Advanced Pick it has been superseded by the **Update Processor** in most areas.

Like similar software on other systems – EDLIN on DOS is another example – the Editor is sometimes called a **line-editor** because it lets you pass, line by line, through an item, at each line allowing you to give commands which will change the contents of that line, delete that line or insert new lines. In contrast, the Update Processor is a **full-screen editor** in which the entire item is displayed at once and you can move around the screen to make changes where you want and in any sequence.

Because of the great power of the Editor to change or delete any item from any file, it is a potentially dangerous tool and should be kept away from non-technical users (and vice versa). It is retained on AP for compatibility with R83 Pick systems.

The sole remaining use of the Editor is probably for making **gang changes** to a file, that is, making the same changes to a large number of items on a file.

The Editor works with two versions of your item: the **original version** and the **new version** which is built up – line by line – as you move through the original version and make your changes.

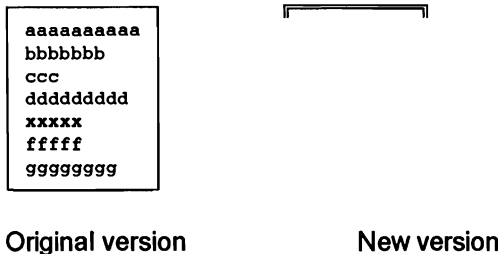
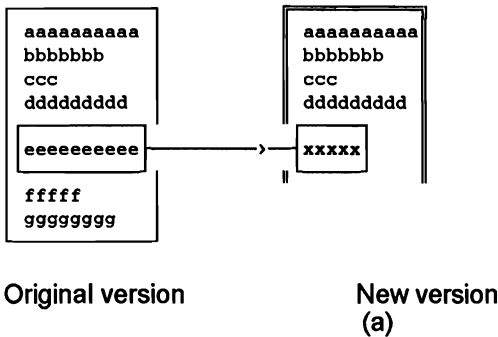
Whenever you display any lines on the screen (possibly by hitting the <Return> key to browse through the item or by using the L command), you are looking at the lines of the **original version**, but when you change a line the amended line (and any earlier lines) are copied to construct the **new version**. As illustrated in Figure 9.1 (a), if we go to line 5 and change

```
eeeeeeeeee to xxxxx
```

then the modified line will be moved across to the new version; the original line in the original version will not be changed. The F command (it stands for **Flip**) will copy any remaining lines from the original version to the new version and then exchange the two versions, as shown in Figure 9.1 (b), so that any further changes will be to the newly amended version, now shown on the left.

Finally, when you file the finished item, any remaining lines will be copied from the original version to the new version and the new version will be written to the file on disk. Unlike some editors which you may have used,

you do not need to move to the end of the item to ensure that all the lines are copied.



(b) After the F (flip) command

Figure 9.1: The Editor

9.1 General points about the Editor

There are a few general points concerning the Editor and the commands which you will use to control the Editor:

- You will give each command in response to the prompt:

.
and each command is executed immediately.

- No record is kept of the Editor commands issued.
- Editor commands are case-insensitive, so that the commands:

DE999
de999

both have the same effect.

- The text within the item is normally case-insensitive, and you may encounter problems if you have issued a TCL CASE-ON / CASE-OFF command.
- The displayed output from command such as:

```
R999/abc/def/  
DE999/A/
```

may be suppressed by putting a *full stop* or a *period* before the command:

```
.R999/abc/def/  
.DE999/A/
```

in which case the lines will *not* be displayed as they are changed.

- The Editor does not keep a backup copy of the original item. When you file your amended item or delete the item, the original contents are lost. A deleted item may be recovered by means of the RECOVER-FD command.

9.2 EDIT command

The general form of the EDIT command is:

```
EDIT file.name {item.list} {(options)}  
OR:  
ED file.name {item.list} {(options)}
```

where *file.name* is the name of the file where the items are held, and *item.list* is a list of one or more item-ids identifying the items to be processed. If the item-list is omitted or * is used, then all the items on the file will be presented for editing.

Some examples are:

```
EDIT MYPROGS PROG01  
will let you edit the item PROG01 which is held on the MYPROGS file.  
If there is no item PROG01 on the file, then a new item will be created.
```

```
EDIT MYFILE ITEM1 ITEM2  
will first present item ITEM1 on the MYFILE file for editing, and, when  
you have made your changes to that item and written it back to the file  
(or deleted it), the Editor will present item ITEM2 on the same file and  
let you edit that.
```

```
EDIT MYPROGS  
EDIT MYPROGS *  
either of these forms will let you edit all the items on the MYPROGS  
file, passing them you to one at a time. Here, as in the previous case,  
each item is processed independently of the others.
```

If you wish to edit just certain selected items, then you may issue an Access SELECT or SSELECT sentence, and – having selected the required items – edit these by means of a command which specifies just the name of the file and without the asterisk, like this:

```
SELECT STOCK WITH MATERIAL "PINE"  
EDIT STOCK
```

```
SSELECT MYPROGS = "[AMEND]"
EDIT MYPROGS
```

If the item-id contains spaces or the special characters " or ' or \ then it must be enclosed in whichever of the special characters " or ' or \ it does not contain, like this:

```
EDIT MYTEXT 'MAJOR PART'
EDIT MYFILE "AA'BB" 'CC"DD' \AA'BB"CC\
```

The Z option will suppress the messages and the beep which sounds when a new item is created.

9.3 Editor commands

The list of the commands which are used to control the Editor is shown in Figure 9.2. We shall look at the most useful here.

number	C	FS	P	T
<Return>	DE	G	PD	TB
?	EX	I	R	U
^	EKK	L	RU	X
A	F	M	S	XF
AS	FD	ME	S?	Z
B	FI	N	SL	

Figure 9.2: Editor commands

Certain commands require a **delimiting character** between the various parameters:

```
DE{n}/string/{p{-q}}
L{n}/string/{p{-q}}
R{n}/string1/string2/{p{-q}}
```

I have used the character / as the delimiter in my examples, but any character (other than the colon and numeric digits) may be used, provided that it does not occur within the string(s) in the command itself.

<RETURN>

Advance the pointer a line at a time.

n

Move the pointer to line number *n*. This is identical to the Gn command.

A

Repeat the last L command.

B

Move the pointer to the bottom of the item.

DE{n}/{string/{p{-q}}}

Delete all those of the next *n* lines – including the current line – which contain the *string* in columns *p* to *q* inclusive.

DE
will delete the current line.

DE7
will delete the next seven lines, that is, the current line and the following 6 lines.

DE5/HELP/
will delete every one of the next five lines which contains the word **HELP**.

DE7/HELP/10-25
will delete every one of the next seven lines which contains the word **HELP** starting anywhere in column 10 to column 25 inclusive.

DE20/END/10
will delete every one of the next 20 lines which contains the word **END** starting anywhere in column 10 or beyond.

DE/PRINT/
will delete the current line if it contains the word **PRINT**.

EX

Abandon the current item and exit from the Editor. If you have made any changes to the item, the Editor will ask you to confirm that you want to abandon it.

EXK

Exit from the Editor and abandon the processing of the current item and any further items in the item-list specified on the EDIT command. If you have made any changes to the current item, the Editor will ask you to confirm that you want to abandon it.

F

Incorporate any changes made and move the pointer to the top of the item, and then continue with the editing of the item. We illustrated this earlier.

FD

Delete the current item from the file and terminate the editing of the item. This command should be treated with respect, because accidental deletion is as permanent as intentional deletion. If you are editing a list of items, then the next item will be presented for processing, otherwise the Editor will terminate.

If an item has been deleted by means of the FD command, it may be recovered by using the RECOVER-FD verb.

FI

File the item: apply any changes made to the item, write the item back to the disk, overwriting the original version if there is one. If you are editing a list of items, then the next item will be presented for processing, otherwise the Editor will terminate.

The special forms:

FI ITEMZ

will save the current item as item ITEMZ on the file which you are using and terminate the sessions, leaving the original item unchanged. If ITEMZ already exists, then the message:

Item is already on file

will be displayed and no action will be taken.

FIO ITEMZ

will save the current item as item ITEMZ on the file which you are using and leave the original item unchanged. If ITEMZ already exists, then it will be overwritten.

FI(MYWORK NEW1

FIO(MYWORK NEW1

will save the current item as item NEW1 on the file MYWORK, leaving the original item unchanged. The FIO form will overwrite NEW1 if it already exists on MYWORK.

FS

Apply any changes made to the item, write the item back to the disk, overwriting the original version if there is one, and **continue with the editing process** – unlike FI which leaves the Editor. The forms:

FS ITEMZ

```
FSO ITEMZ
FS(MYWORK NEW1
FSO(MYWORK NEW1
```

are available and behave much like the FI forms described earlier.

Gn

Move the pointer to line number *n*. This is identical to entering just the number *n*.

I

Insert one or more lines after the current line.

The Editor will display a prompt of the form:

00n+

to show that lines are being inserted after line number *n*. You can then enter the new information, line by line, each line being prompted by the same message:

00n+

An input of null (just hit <Return>) will terminate the insert action.

Care must be taken when you want to insert blank lines. This may be achieved by inserting lines containing a dummy character-string which is then replaced by null, or by inserting a line containing attribute-marks, as illustrated below.

I xxxxx

Insert the single line of data xxxxx after the current line. There is a space between the I and the data (otherwise the command will be rejected). For example:

I PRINT "ENTER NEW ADDRESS"

will insert the line

PRINT "ENTER NEW ADDRESS"

immediately after the current line. The command:

I ^^<Return>

where the ^ represents the attribute-mark (entered by the sequence <Ctrl> ^ character), will insert **three** null lines.

Ln

Display n lines starting at the line following the current line. For example, if the current line is line 1, then the command:

L20
will display lines 2 to 21, inclusive.

L{n}/string/{p{-q}}

Display all those of the next n lines which contain the *string* in columns p to q inclusive. For example:

L20/AREA/
will display those of the next 20 lines which contain the word AREA in any position in the line.

L5/AREA/6-12
will display those of the next 5 lines which contain the word AREA within columns 6 to 12 of the line.

L/AREA/
will scan the item and display the first line which contains the word AREA anywhere on the line.

L9999/AREA/
will display all the lines (from here on down) which contain the word AREA anywhere.

L{n}:xxxxx

Scan the next n lines of the item and display any which begin with the characters xxxxx. The command:

L20:\
will display all of the next 20 lines which start with the \ character.

L:1000
could be used to locate the statement label 1000 in a Basic program. Any occurrences of 1000 other than in columns 1 to 4 would be ignored.

ME{n}/{iiii}/{m}

Merge, or copy, n lines from item *iiii* on the same file as the current item, starting at line number m , and insert them after the current line in the item being edited. If you omit n , a value of 1 is used; if you omit m , a value of 1 is used; if you omit *iiii*, then the current item is used. The command:

ME5/ANDATA/10
will copy the five lines from the item ANDATA, starting with line 10 (that is, lines 10, 11, 12, 13, and 14) and insert them after the current line in the item which you are editing. The item ANDATA is assumed to be on the same file as the item which is being edited, and will itself be unchanged.

ME3//20

will copy the three lines 20, 21, and 22 from the current version of the item which you are editing and insert them after the current line. This duplicates these lines.

If the item is to be taken from another file, then you will use the form:

ME{n}{fffff {iiii}}{m}

Merge, or copy, *n* lines from item *iiii* on the file *ffff*, starting at line number *m*, and insert them after the current line in the item being edited. The command:

ME3(ALLPROGS UPDP)7

will copy the three lines 7, 8 and 9 from the item UPDP on the file ALLPROGS.

ME9999(SOURCE MASTER)

will copy all the lines from item MASTER on the file SOURCE.

ME9999(MAINPROGS

will copy all lines from the item on the file MAINPROGS which has the same item-id as the item being edited.

P

Display the next 23 lines of the item. This is a **prestored L23** command which is automatically set up on entry to the Editor. We shall discuss **prestore commands** later.

R{n}

Replace the next *n* lines, including the current line. This is much like the I command and invites you to type in *n* lines of information to replace the next *n* lines of the item. When you enter a null line, this will terminate the replacement activity.

R{n}/string1/string2/{p{-q}}

Replace the string *string1* by *string2* for the first (left-most) occurrence in columns *p* to *q* inclusive of the next *n* lines starting at the current line.

Note the following points about this form of the R command. Some of these are illustrated by the examples shown below:

- If you omit the number of lines, then a value of 1 is assumed.
- The strings may be of different lengths
- Either string may be null
- Unless you use the RU form described below, only the first (left-most) occurrence of *string1* will be changed.
- As mentioned above, my examples use the / character to separate the

strings, although any character (other than the colon and numeric digits) may be used.

- If the character ^ is used within *string1*, it has a *wild card* effect and represents *any* character. However, the character ^ within *string2* is respected as the character ^
- The ^ command may be used to override this wild card effect.
- An attribute-mark, that is, <Ctrl> and ^ character, within *string2* will *truncate* the line at that point.

Here are some examples:

R/WAGE/SALARY/
will replace the first occurrence of the word WAGE by the word SALARY in the current line.

R5/WAGE/SALARY/
will replace the first occurrence of the word WAGE by the word SALARY in the next five lines, starting with the current line.

R/WAGE//
will remove the first occurrence of the word WAGE from the current line.

R/ /WAGE/
will append the word WAGE to the right-hand end of the current line. The string of spaces (*string1*) must be longer than any string of spaces which may occur within the line. There is no APPEND command.

R//WAGES/
will insert the word WAGE at the left-hand end (that is, the beginning) of the line.

R/WAGE/SALARY/30-50
will replace the first occurrence of the word WAGE in columns 30 to 50 inclusive by the word SALARY in the current line.

R5/WAGE/SALARY/30-50
will replace the first occurrence of the word WAGE in columns 30 to 50 inclusive by the word SALARY in the next five lines, starting with the current line.

R/WAGE/SALARY/30
will replace the first occurrence of the word WAGE in columns 30 onwards by the word SALARY in the current line.

R/*/-/30-30
will replace any * in column 30 by the - character. Any * elsewhere in the line will be ignored.

R5/WAGE/SALARY/30
will replace the first occurrence of the word WAGE in columns 30

onwards by the word SALARY in the next five lines, starting with the current line.

R999/^//30

will remove the 30th character from this and the next 998 lines. The ^ is the character over the figure 6 on the keyboard and is the Editor wild card character.

RU{n}/string1/string2/{p{-q}}

This is similar to the R command described above, except that the RU command will replace *all occurrences of string1 by string2* in columns *p* to *q* inclusive of the next *n* lines starting at the current line. Without the U (for universal), the R command replaces only the first, left-most occurrence of *string1*.

RU5/WAGE/SALARY/

will replace all occurrences of the word WAGE by the word SALARY in the next five lines, starting with the current line.

RU999/^//30-50

will remove all the characters from columns 30 to 50 inclusive from this and the next 998 lines.

T

Move the pointer back to the top of the item. Any changes which have been made will not be visible until an F command or FS command is issued.

X

Undo the effect of the last ME or R or I command.

9.4 Prestore commands

The Editor command:

P

will display a screen-full of lines, exactly like the command:

L23

This happens because the command L23 is **prestored** when the Editor is invoked.

Up to ten prestored commands may be held at one time; each is identified by its store-number: P0, P1 through to P9 (P and P0 are equivalent). When you start to use the Editor, prestores P1 to P9 are empty.

If you are using an Editor command – or the same sequence of Editor

commands – over and over again, then you may store the command(s) and recall them when required. For example, we might store a command such as:

```
R/!/. /
```

in prestore number 6 by entering the command:

```
P6 R/!/. /
```

and then invoke this by entering:

```
P6
```

All ten prestores are used in the same manner, and there is no distinction between them. You can check the current prestores by means of the command:

```
PD
```

Let us imagine that we wish to change the word WAGE to the word SALARY in line 7 of an item, then we might store the commands:

```
P1 G7
P2 R/WAGES/SALARY/
```

and then every time you issue the command:

```
P1 will move the pointer to line 7 of the item, and
```

```
P2 will change the word WAGE to the word SALARY
```

It is possible to store a *sequence of several commands* in one prestore. This is done by separating the individual commands by the <Esc> character, shown here as the [character. Thus, we might combine the effect of these two commands by the single command:

```
P1 G7[R/WAGES/SALARY/
```

and when we enter the command:

```
P1
```

the Editor will move to line 7 and change the word WAGES to SALARY.

It is possible to **program** the Editor and construct a *loop* of commands in which the final command in the sequence recalls the prestore command itself. For example:

```
P1 R999/WAGES/SALARY/[FI[P1
```

Prestore commands such as this are useful when making *gang changes* to a number of items, a process which otherwise might need a complex program to scan and change the items.

This is illustrated by the sequences in Figure 9.3 (a) and (b) either of which will change the word PERSONNEL to P in line 3 and/or the word ADMINISTRATION to A in line 7 of all items on the PAYROLL file.

```
:EDIT PAYROLL *
.P1 G3[R/PERSONNEL/P/
.P2 G7[R/ADMINISTRATION/A/[FI
.P3 P1[P2[P3
.P3
```

(a)

```
:EDIT PAYROLL *
.P1 G3[R/PERSONNEL/P/[G7[R/ADMINISTRATION/A/[FI[P1
```

(b)

```
:EDIT PAYROLL *
.P1 G3[R/PERSONNEL/P/[G7[R/ADMINISTRATION/A/[F[P1
.P1
```

(c) A loop error

Figure 9.3: Prestored commands

You must take care that your prestore loops terminate. If you had accidentally issued the commands in Figure 9.3 (c), using the F command instead of FI, then this would keep processing the same item indefinitely, or until you terminate the process by means of the <Break> key.

9.5 RECOVER-FD

If an item has been deleted by means of the Editor FD command, it may be recovered by issuing the TCL command:

```
RECOVER-FD
```

immediately after the Editor has terminated. The RECOVER-FD verb displays the message:

```
RECOVER "ffff iiii" (Y/N)?
```

and if you enter Y, the item will be returned to the file in the state that it was when you last started to edit it.

RECOVER-FD is a TCL verb and the recovery must be done immediately after the deletion. If any other activity is performed before the RECOVER-FD is issued, then the lost item may be irretrievable. For this reason, it is not possible to recover any other than the last item which deleted when processing a list of items, as with:

```
EDIT INVOICES *
```

RECOVER-FD can also be used to recover the *last item* which was deleted by the Update Processor ^XO sequence.

9.6 Summary

The Pick Editor is an alternative to the Update Processor and can be used to create / amend / delete any item on any file.

Within Advanced Pick, the main use of the Editor is probably for carrying out program-controlled editing and for performing gang changes, that is, making the same changes to a large number of items on a file.

10

The Update Processor

The Update Processor is a major tool of the Advanced Pick system. It performs two roles:

- It is a **general purpose screen-editor**, and
- It is a **dictionary-driven, file maintenance utility**.

As a general purpose screen-editor, the Update Processor allows the technical user to create / maintain / delete any item on any file and supersedes the Pick Editor which we discussed in Chapter 9. In this context, the Update Processor might be used to handle:

- Output Processor documents,
- Basic source programs,
- Macros,
- Menus,
- Procs,
- Runoff documents,
- Access phrases,

and so on.

As a dictionary-driven, file maintenance utility, the Update Processor allows you to create / maintain / delete any item on any file and is a viable alternative to bespoke file maintenance programs within an application system. The attribute definitions used by the Update Processor are identical to regular Access attribute definitions, but they may be extended to enable the Systems Analyst to impose checks and constraints to control the data which users enter when using the Update Processor to maintain files and items.

There is a special UD macro which invokes the Update Processor for handling items such as D-pointers and attribute definitions which are held on the DICT section of a file.

10.1 UPDATE command

The Update Processor is invoked by a command of the form:

```
UPDATE file.name {item.list} {(options)}
```

or by the equivalent abbreviated forms:

```
UP file.name {item.list} {(options)}  
U file.name {item.list} {(options)}
```

where *file.name* identifies the file which is to be maintained, and *item.list*

is the list of item-ids of the items to be processed. We shall use the full form – UPDATE – in our examples.

The options available with the UPDATE verb are shown in Figure 10.1.

- c to clear the screen before each item is presented.
- I to allow the item-id field to be amended. By changing the item-id, you can leave the current item and move to another item on the file. This is identical to the action of the ID-PROMPT modifier.
- L to indicate that the processing is to be **look-only** and items cannot be updated/field. If you attempt to use a sequence such as ^XF to file a *look-only* item, the Update Processor will reject this with a message:

Unable to modify 'xx', updated with look-only option

The ^XX or ^XK sequences can be used to leave a *look-only* item.

- R to edit the raw data. This allows <Esc> characters to be handled; without this option, any <Esc> characters are converted to spaces.
- s to disable the spelling checker on entry to the Update Processor.
- w to invoke the GraPICK presentation of the Update Processor.

Figure 10.1: Update Processor options

10.2 Some examples of the UPDATE command

A typical use of the Update Processor might be:

```
UPDATE STOCK '1000' DESCRIPTION PRICE
UPDATE STOCK 1000 DESCRIPTION
```

When such a command is processed, the Update Processor will display a screen image showing the **item-id**, the **description** and the **price** of item 1000 on the STOCK file, allowing you to amend the two data fields:

```
STOCK    1000
DESCRIPTION  DESK, GREEN-BLUE, ASH
```

PRICE	56.00

All movement of the cursor through the screen image, data amendment and filing the finished item is performed by means of the Update Processor <Ctrl> keys. The identity, nature and appearance of the data is controlled by the attribute definitions for the datanames DESCRIPTION and PRICE.

If no datanames are specified on the UPDATE command:

```
UPDATE STOCK '1000'
```

then the Update Processor will default to a list of datanames which are specified in the macro field (attributes 15 and 16) of the data-level identifier item for the STOCK file. If there is no such macro list, then the entire item will be displayed and the Update Processor will behave exactly like a full-screen editor; this is the situation when editing an item such as an Output Processor document or a Basic program.

If there is a macro for the file and you do not wish to use the datanames defined there but wish to edit the item directly, then you can use the keyword ONLY:

```
UPDATE ONLY STOCK '1000'
```

to suppress the macro list, as in Access sentences.

Commands such as:

```
UPDATE STOCK '1000' '2345' '3456'  
UPDATE STOCK 1000 2345 3456
```

will apply the Update Processor to each of the specified items in turn. As one item is finished, by means of one of the ^X sequences, the next item will be presented. In this situation, you may use the sequences:

- ^XF to file the item,
- ^XX to abandon the item,
- ^XB to return to the previous item in the list,
- ^XK to abandon the item and any remaining items in the item-list.

Forms such as:

```
UPDATE STOCK *  
UPDATE STOCK * DESCRIPTION PRICE
```

will apply the Update Processor to *all* existing items on the file.

The forms:

```
UPDATE STOCK
```

```
UPDATE STOCK DESCRIPTION PRICE
```

without any item-ids, will *create a new item with a unique item-id* of the form:

```
dddx
```

and based upon the current internal date (*dddd*) and a system-wide sequence number (*x*), and then proceed to accept the data for that item. When the item is filed (by the ^XF sequence), control will return to TCL. The ID processing code may be used to control the form of the new item-ids which are created.

10.3 DATA-ENTRY

An extension of this form uses the DATA-ENTRY keyword:

```
UPDATE STOCK DATA-ENTRY
UPDATE STOCK DESCRIPTION PRICE DATA-ENTRY
```

to create a new item with a unique item-id, as in the previous case, but as the final field (the PRICE in this instance) is entered, each item is automatically filed and a new item will be presented. The sequence ^XK will terminate the activity and return to TCL.

10.4 ID-PROMPT

If the ID-PROMPT keyword or the I option is used:

```
UPDATE STOCK '1000' ID-PROMPT
UPDATE STOCK '1000' '2000' (I
UPDATE STOCK ID-PROMPT
UPDATE STOCK (I
```

the Update Processor will ask for the item-id (if no item-list is included in the command), and then display the item-id just like any other attribute and allow the user to amend it, thereby moving to another item.

The Update Processor will accept the file-name, the list of item-ids, and the datanames in almost any sequence:

```
UPDATE file.name item.list datanames
UPDATE file.name datanames item.list
UPDATE item.list file.name datanames
```

provided that this is not ambiguous. The syntactic **similarities** and the semantic **differences** in the forms:

```
UPDATE STOCK 1000 2345
UPDATE STOCK DESCRIPTION 2345
UPDATE STOCK DESCRIPTION QUANTITY
```

means that you must pay particular attention to the datanames and the item-ids which you use in your commands. To avoid any ambiguity, you should enclose the item-ids in apostrophes. If the Update Processor

cannot resolve any ambiguity, it will abandon the process with a message of the form:

[101] Duplicate definitions for attribute: n

10.5 The UD command

The UD macro invokes the Update Processor to create and maintain **attribute definitions** held on the DICT section of a file. It is also used to maintain **D-pointers** and **Q-pointers** held on the MD and on the DICT section of files.

Here are some examples:

- UD STOCK QUANTITY PRICE
create or modify the **attribute definitions** for the datanames QUANTITY and PRICE which are held on the DICT section of the STOCK file.
- UD STOCK STOCK
change the **data-level identifier D-pointer** for the data section which is held on the dictionary section of the STOCK file.
- UD MD STOCK
change the **file definition D-pointer** for the DICT section of the STOCK file which is held on the MD.

In these forms, the Update Processor is invoked with a standard list of attribute definition fields to determine which fields are presented. Figure 10.2 shows the fields used by the UD verb, depending upon which type of item is being maintained.

dictionary-code	correlative	reallocation
base	attribute-type	hotkey.all
modulo	column-width	hotkey1
structure	input-conversion	:
retrieval-lock	macro	hotkey9
update-lock	output-macro	hotkey0
output-conversion	description	

(a) D-pointers

dictionary-code	attribute-type
account	column-width
filename	

(b) A Q-pointer items

dictionary-code	attribute-type	hotkey.all
attribute-count	column-width	hotkey1
substitute-header	input-conversion	:
structure	macro	hotkey9
output-conversion	output-macro	hotkey0
correlative	description	

(c) Attribute definitions

Figure 10.2: Fields used by the UD macro

For other items – such as data items – on the DICT section of a file, no standard list is used.

The UD verb displays a screen comprising the fields from the lists in Figure 10.2, and the contents of the items are created and maintained in the normal manner. We discuss the form and function of these fields later.

Using the UD verb to create and maintain your Access attribute definitions rather than the standard line EDITor, has the advantage that it displays the name of each field as a prompt for each part of the definition, and that the ? help facility can also be used. Indeed, D-pointers may only be amended by the UD verb and cannot be edited by means of the standard EDITor.

10.6 D-pointers and the Update Processor

When we looked at Figure 4.36 during our discussion of D-pointers, we saw that file definitions contain a number of fields which are specifically used with the Update Processor:

- Attribute 7: The **correlative** can hold any of a number of processing codes to control features such as indexing, bridging, and item-id format, which apply to the whole file and not to just a single field. We look at processing codes in Chapter 14.
- Attribute 8: The **conversion** or **output conversion** can hold any of a number of processing codes to control the format of the data as it is displayed by the Update Processor.
- Attribute 9: The **format** and **justification** of the displayed item-ids.
- Attribute 10: The **column width** for the displayed item-ids on that file.
- Attribute 14: This specifies any processing codes which are to be used for **input conversion**.
- Attribute 15: This specifies the **macro**, a default output list of datanames which is to be used if this file is handled without an output list in an UPDATE command.
- Attribute 16: This specifies the **output macro**, a default output list of datanames which are to be shown if this file is used without an output list in an Access sentence.
- Attribute 17: This is the **description**, a free-format text field for any comments concerning the usage and nature of the file, which is to be displayed if the user enters ? when inputting the item-id, as with the I option or the ID-PROMPT modifier.
- Attributes 20 to 30: These contain CALLs to any subroutines which are to be invoked by the **hotkey** sequences when handling items on this file. We discuss hotkeys in a moment.

The UD macro allows you to change any of these fields and verifies the consistency of your changes.

10.7 Attribute definitions and the Update Processor

Attribute definitions are used by the Access enquiry language and also by the Update processor. They provide information to control the format and appearance of the data when the dataname is used in an Access sentence, and they hold information which controls and validates the data when the dataname is handled by the Update Processor.

The fields which make up an attribute definition are illustrated in Figure 10.3. We shall look more closely at the contents of attributes 1 to 10 when we discuss the structure and use of attribute definitions with Access in Chapter 13.

- Attribute 0: the item-id of an attribute definition is the **dataname** which will be used in Access sentences and UPDATE commands.
- Attribute 1: the **dictionary-code** is A to identify this as an attribute definition.
- Attribute 2: the **attribute-count** indicates which attribute of the data items is controlled by this dataname.
- Attribute 3: the **substitute-header** is the text heading which is to be shown with this field. If this is null, the dataname itself is used.
- Attribute 4: the **structure** can be used to indicate that this field is related to others when producing Access reports.
- Attribute 7: the **output-conversion** is any processing code(s) which are to be applied to the data as it is displayed. This is used for inserting decimal points, commas and currency symbol, for converting dates and times to the required format.
- Attribute 8: the **correlative** is any processing code(s) which indicate how the data is to be assembled from the data in the items.
- Attribute 9: the **attribute-type** specifies the left or right justification of the displayed data.
- Attribute 10: the **column-width** indicates the minimum width of the column when this field appears on an Access report.
- Attribute 14: the **input-conversion** specifies any conversion code(s) which are to be applied immediately after this field has been accepted

00	dataname
01	dictionary-code
02	attribute-count
03	substitute-header
04	structure
05	not used
06	not used
07	output-conversion
08	correlative
09	attribute-type
10	column-width
11	not used
12	not used
13	not used
14	input-conversion
15	macro
16	not used
17	description
18	not used
19	not used
20	hotkey.all
21	hotkey1
30	hotkey0

Fig 10.3: Attribute defn

by the Update Processor.

- Attribute 15: the **macro** specifies a list of datanames which are to be used when zooming to another item with the Update Processor.
- Attribute 17: This is the **description**, a free-format text field for any comments concerning the usage and nature of the field, which is to be displayed if the user enters ? when inputting this field.
- Attributes 20 to 30: These contain the CALLs to any subroutines which are to be invoked by the **hotkey** sequences as this field is being handled by the Update Processor. We discuss hotkeys later.

The UD macro validates the changes which you make to your attribute definitions.

10.8 UP versus application programs

The Update Processor has many attractions for the Systems Analyst who is designing an application system:

- 1) It is dictionary-driven.
- 2) Access datanames and attribute definitions can be used, and these may be extended for more detailed control of the way in which a field is handled by the Update Processor.
- 3) Any changes to the attribute definitions for a file are immediately implemented when the Update Processor is used to update those data fields on the file.
- 4) Very little – if any – bespoke code need be written to produce a viable maintenance utility.

There are several points to be recognised if you are thinking about using the Update Processor as a part of your application system:

- Your users must master some of the <Ctrl> keys and key sequences which control the Update Processor. At the very least, these will include:

^XF to file the item

and the cursor-movement keys:

^K
^J
^N
^B
^T

- A mastery of the full range of <Ctrl> keys may be beyond the grasp of many non-technical and/or infrequent users.
- The advanced features and functions such as:
 - Zooming,
 - Cruising,

- Double-clutching,
are best reserved for more experienced users. We look at these later.
- The speller should be switched on only when absolutely necessary to the application.
- The programmers must be conversant with the form and content of attribute definitions in order to exploit the features of the Update Processor. Traditionally, many programmers have concentrated on programming and ignored all but the most elementary features of attribute definitions.
- It may be necessary to produce a number of Basic subroutines to validate the data and control the actions performed by the users within the Update Processor.

In general, it may be more attractive to use the Update Processor for simpler *ad hoc* file updates, whilst retaining Basic programs for the more complex tasks of creating new items and for performing detailed calculations, data validation and data manipulation. In all cases, it is better to provide your users with a series of macros (or Procs) which will invoke the Update Processor to handle specific fields on a specific file, rather than giving them free rein to use the Update Processor to change any fields on any items on any files.

M
UPDATE file.name item.list datanames
PQ
HUPDATE file.name item.list datanames
P
EXECUTE 'UPDATE file.name item.list datanames'

Figure 10.4: Invoking the UP

10.9 Using the Update Processor in a Proc / macro / menu

It is possible to invoke the Update Processor from a macro, a menu, a Proc or a Basic program, as illustrated in Figure 10.4.

If you are feeding a command, then this may be specified as a multivalue and without the <Ctrl> keys; for example, the macro line:

```
UPDATE file item]XO
```

will delete the item. If you are feeding a data string, then this will be enclosed in quotation marks; for example, the macro line:

```
UPDATE file item]"AAAAAAAAA"
```

will set the first attribute of the item to AAAAAAAAAA and then await further commands from the user.

A prestore can be employed in making gang changes – that is, to make the same changes to several items on a file. We illustrate this in Figure 10.5 by looking at a sequence which will change the word YES to NO on attribute 7 of all the items on the STAFF file. If you were using the Pick Editor, you might use a sequence of commands like that in Figure 10.5 (a). The equivalent Update Processor sequence is shown in (b).

```
EDIT STAFF *
P1 7]R/YES/NO/]FI]P1
P1
```

(a) Editor commands

```
U STAFF *
<Ctrl>ZL                                     <== Load the prestore buffer
Z'7'MA'YES'R'NO'MRXXFP<Return>
<Ctrl>P                                     <== Invoke the prestore buffer
```

(b) Update Processor commands

Figure 10.5: Making gang changes

For tasks which are to be performed frequently, it may be possible for the prestored command to be constructed and saved on a file and then called in as the Update Processor is invoked.

10.10 The Update Processor and Basic programs

The Update Processor has facilities specifically for use during the development of Basic programs and subroutines:

- o The sequence ^XR will file, compile and run a program item. This invokes the equivalent COMPILE-RUN command. ^XR can also be used to file and invoke a macro, a menu or a Proc.
- o The sequence ^XC will file, compile and catalogue a program/subroutine item. This invokes the equivalent COMPILE-CATALOG command.

Note that these use the COMPILE (case-insensitive) compiler. There is no equivalent form which will use the BASIC compiler.

10.11 Living with the Update Processor

The Update Processor is controlled by the Advanced Pick <Ctrl> keys and sequences discussed in Chapter 7. These allow you to:

- o Move the cursor through the data in the item,
- o Change and/or delete the existing data,
- o Insert new data into the item.

When using the Update Processor as a text editor, there are additional

facilities for:

- Performing search and replace activities,
- Cutting and pasting,
- Saving sequences of commands to make identical changes to a number of items,
- Spelling checking,

which are of great value for general editing – documents and text, Basic programs, and such – but which may not be appropriate for maintaining data items. We shall look at these in a moment.

10.11.1 Entering data

There are some general points which are of interest when entering data via the Update Processor:

- In general, data is entered directly into a field and existing data may be changed by means of the editing <Ctrl> keys.
- When you type in new data for a numeric field or a date field, the existing data may be cleared, depending upon whether or not there are processing codes in the attribute definition for that dataname.

A simple text field – such as a description or an address – with no processing codes, will not be cleared and you may move the cursor through the text and overwrite / insert new text into the existing data. However, if the attribute definition uses a processing code – such as a date or a time – then the field will be cleared immediately you enter any new data into the field; if the attribute definition includes an input conversion code CU, then the data may be edited character by character.

- If a field represents a date and has a D input conversion code, then you may enter the date in any of the acceptable formats:

```
dd/mm/yy
dd mmm yyyy
mmm dd yyyy
```

and the following special forms:

/ to indicate that today's date is to be used.

d to denote day *d* of the month.

d/m to denote day *d* of month *m* of the year (if your date is set to the UK format), or day *m* of month *d* (if your date is set to the US format).

ddd to denote Julian date *ddd* of the year.

yy/ddd to denote Julian date *ddd* of year *yy*.

- If a field represents a clock time and has an MT input conversion code, then you may enter:

/ to indicate that the current time is to be used.

- o You may enter: ,
?

to display a help screen about the particular field. This help message is held in the description of the attribute definition for that dataname.

- o The ^V key is used to enter a new value to a set of multivalues.
- o If a field comprises a set of multivalues, each multivalue is displayed on a separate (unlabelled) line.
- o If the system beeps when you attempt to add a new multivalue to an attribute, this means that a limit has been placed on the number of multivalues which may be held in that attribute and no further multivalues may be added. The V processing code is used to impose such a limit.

10.11.2 Changing data

The full range of <Ctrl> keys and sequences which are used with the Update Processor are shown at the back of this book. The most important control sequences for editing the text of an item are:

- ^E to clear / delete to the end of the current attribute.
- ^H and the equivalent <Backspace> key to overwrite the character to the left with a space.
- ^L to delete the current character.
- ^O to delete to the end of the current word.
- ^R to toggle between the otype and insert modes. When this key is pressed, the data will be broken at the cursor position, allowing new data to be inserted at that point. Pressing the ^R key a second time will join the broken text. The letter O or I is displayed in the top-right corner of the screen to indicate whether the Update Processor is in the Otype mode or in the Insert mode.
- ^V to insert a value-mark.
- ^W to insert one space.
- ^ZZ to undo the last change made.

10.11.3 Seeking help

Attribute 17 of D-pointer items and attribute definitions holds the **description** for that dataname. This is a free text field and is provided to hold any comments concerning the use and nature of the dataname. This text is also used as a **help message** displayed if you enter:

?

when using the Update Processor to handle a dataname. If there is no description for a dataname, no action is taken when you enter the ? request.

10.11.4 Default values

No default values are assumed for any attributes processed by the Update Processor, and there is currently no processing code which will impose a default value. However, the situation can be handled by using a feature such as a **hotkey** to call a Basic subroutine in which you reset:

- ACCESS(10) to generate an item-id for the current item, or
- One of the attributes of ACCESS(3) to assign a specific attribute. ACCESS(3) contains a copy of the current item.

to provide a default value.

10.11.5 Handling multivalues

Several Update Processor facilities are related to **multivalued** attributes. The following points are of interest:

- The ^V key will allow you to enter a value-mark and multivalues. Thus, the sequence of keys:

```
aaaaa^Vbbbbbb^Vccccc^Vdddd
```

will appear as:

```
aaaaa
bbbbbb
ccccc
dddd
```

each multivalue being shown on a separate line of the display.

- There are no facilities for entering **subvalues** via the Update Processor. If you need to enter subvalues, you must use the standard EDITor and enter these using the familiar <Ctrl>\ key to enter the separating subvalue-mark.

Certain processing codes are specifically associated with multivalues and the Update Processor:

- The O code specifies that multivalues are to be sorted into left-justified ascending order when they have been added to the attribute.
- The V code specifies a maximum number of multivalues which are allowed in the attribute. The system will beep if you attempt to enter more than the specified number of multivalues.

We shall look at these and the other processing codes later.

10.12 Special UP facilities

There are a number of special facilities offered by the Update Processor. When the Update Processor is used to update files, then:

- Bridges,
- Zooming,
- Cruising,
- Double-clutching,

will be of interest, and when editing general items – such as text, macros, menus and programs, then:

- Prestored commands,
- Spelling checking,
- Cut and paste,
- Search and replace,

will be of use. Let's look at these.

CLIENT	
000 1201	<== Client number
001 JONES BROS	
002 34 HIGH STREET	
003 234]345]567	<== Order numbers
ORDER	
000 234	<== Order number
001 1201	<== Client number
002 8166	
003 25000	

Figure 10.6: Bridging between files

10.12.1 Bridges

Let us imagine that our application uses two files: CLIENT and ORDER – and that each CLIENT item holds the numbers of the orders submitted by that client, and that each ORDER item holds the number of the client who placed the order. This is illustrated by the items shown in Figure 10.6.

The concept of **referential integrity** between the data on our files demands that, if the CLIENT item for client 1234 shows that this client submitted order number 234, then the ORDER item 234 must also indicate that this was submitted by client number 1201, and vice versa. Furthermore, we cannot remove order number 234 from the CLIENT item whilst ORDER item 234 exists, and we cannot remove client 1201 from the ORDER item whilst CLIENT item 1201 exists. There is an implied (logical) bridge between the order-number in attribute 3 of the CLIENT file and attribute 0 (the item-id) of the ORDER file, and (in the opposite direction) between the client-number in attribute 1 of the ORDER file and attribute 0 of the CLIENT file: for each client number in the ORDER items, there must be a corresponding client on the CLIENT file, and for each order number in the CLIENT items, there must be a corresponding order on the ORDER file.

Most database management systems require considerable programming effort to enforce such referential integrity between the files. Advanced Pick controls referential integrity fairly simply by the concept of **bridging**. The B processing code will allow us to specify such a bridging relationship between the files. We would use the correlative:

```
BCLIENT;1;3
```

in the data-level identifier item of the ORDER file to establish such a bridge to operate in both directions, from the ORDER file to the CLIENT and from the CLIENT file to the ORDER.

Whenever an item is added to or deleted from the ORDER file, attribute 3 of the CLIENT file is changed accordingly. Furthermore, in this situation, the Update Processor will not allow the order-numbers on the CLIENT file to be deleted until the corresponding order on the ORDER file has been deleted. We discuss the B processing code in Chapter 14.

10.12.2 Cruising

A major enhancement of Advanced Pick over R83 Pick systems is the ability to create and use one or more secondary indexes for a file. Thus, although the items on, say, the SUPPLIER file are uniquely identified by the supplier-number, we can construct an index on, say, the suppliers' names which will enable us to seek a certain name and use this to obtain the item-id (or item-ids) of the supplier(s) with that name; we can then use this item-id in the normal manner to retrieve the entire item. There may be any number of such indexes for a file, and each may be based upon data extracted from the item in any manner. For example, we may have chosen to index the supplier's telephone number, the name and telephone number, the postcode, and so on. We discuss indexes in Chapter 15.

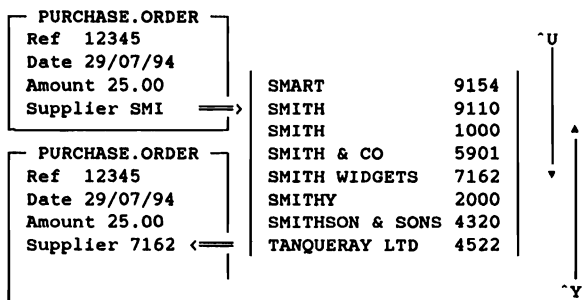


Figure 10.7: Cruising

If an index is available for an attribute which is being handled by the Update Processor, you may cruise through this index looking for the required value. Let us illustrate this by considering the following scenario:

- You are using the Update Processor to maintain a purchase order item on the PURCHASE.ORDER file, and ...

- You are required to enter a supplier code. This will be held as a data field within the PURCHASE.ORDER item and is also the item-id of the appropriate supplier item on the SUPPLIER file.
- You know the supplier's name – SMITH – but not the supplier code, but ...
- An index is available for the SUPPLIER file based upon the supplier's name.

Now, if – instead of a supplier code – you enter just the first few letters of the supplier's name, such as SMITH or SMI or even S followed by the:

^U

key, then all the names in the index starting with SMITH – SMITH, SMITH & CO, and so on – will be presented one by one, as the ^U key is repeatedly pressed. The:

^Y

key will take you back up the index. When the required name is presented, you will press:

<Return>

or:

^M

and that supplier code will be placed into the field which is being updated. This concept, known as **cruising**, is illustrated in Figure 10.7.

As we shall see later, these actions require that certain indexes and pointers are created to link the files in this way.

10.12.3 Zooming

A further concept which is of value when two files are associated in this manner is that of **zooming**. Zooming allows you to select an attribute in the current item and use this as an item-id to go to the associated file and process the corresponding item there.

Let us imagine that you are modifying an item on the ORDER file and that the cursor is positioned at the CLIENT field where the client code is 8901, the key to an item on the CLIENT file.

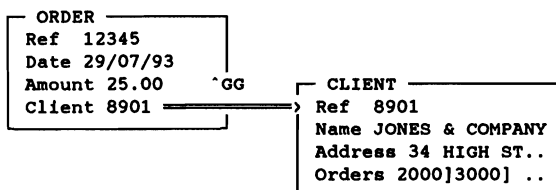


Figure 10.8: Zooming

If you press the sequence:

`^GG`

the Update Processor will jump across to item 8901 on the CLIENT file, where you can amend that CLIENT item. When you use a `^X` sequence to file and terminate the activity on the CLIENT file, you will return to the ORDER item. This is illustrated in Figure 10.8. As we shall see later, these actions require that certain indexes and pointers are created to link the files in this way.

10.12.4 Double-clutching

Let us imagine that you are using the Update Processor to modify item 3000 on the ORDER file, as illustrated in Figure 10.9. The concept of **double-clutching** combines the actions of zooming and cruising to allow you to use the current CLIENT field 8901 to pass to that item on the CLIENT file, and then to browse through the items on the ORDER file for that client.

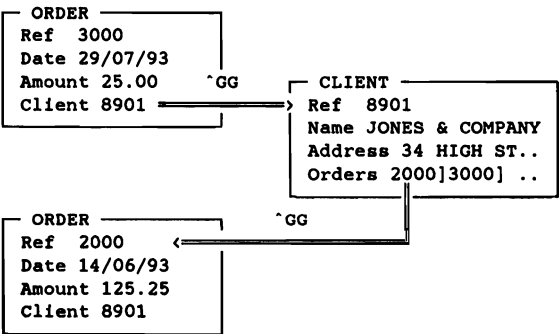


Figure 10.9: Double-clutching

The sequence `^GG` is used to make both transfers. This zooming from one item to another may continue virtually indefinitely.

At each stage, you may make any changes to the item which is currently active, leaving (and returning to the previously active item) by means of a `^XF` or `^XX` sequence.

If you use this facility recursively and attempt to change an item which you (or another user) is already editing, such as item 3000 in this illustration, then the system will report:

`'3000' locked by line 10 retry/look/quit (r/l/q)?`

As we shall see, these actions require that certain indexes and pointers are created to link the files in this way.

1000	9007
01 SMITH & SONS	01 9630
02 ANSTEE PARK	02 1001
03 9009	03 12345
1001	9008
01 SMYTHE INDUSTRIES	01 9632
02 34 HIGH STREET	02 2000
03 9007	03 34575
2000	9009
01 WATKINS LTD	01 9635
02 123 ELLAND RD	02 1000
03 9008]9010	03 40025
3000	9010
01 LANDAU & SONS	01 9639
02 P.O.BOX 43	02 2000
	03 4550

Figure 10.10 The data sections of the files

10.12.5 Cruising, zooming and double-clutching

To illustrate some of the features of the Update Processor, including the use of bridges, cruising, zooming and double-clutching, let us look at two files: the PURCHASE.ORDER file and the SUPPLIER file. Figure 10.10 shows the data items on the two files, and Figure 10.11 shows the important parts of the items on the DICT sections of the two files.

DICT of the SUPPLIER file		
SUPPLIER		
dictionary-code	D	
correlative	I9876A1	(1)
macro	NAME ADDRESS PO.NUMBER	(2)
NAME		
dictionary-code	A	
attribute-count	1	
ADDRESS		
dictionary-code	A	
attribute-count	2	
PO.NUMBER		
dictionary-code	A	
attribute-count	3	
input-conversion	IPURCHASE.ORDER;A0	(3)
DICT of the PURCHASE.ORDER file		
PURCHASE.ORDER		
dictionary-code	D	
correlative	BSUPPLIER;2;3	(4)
	I8765A0	(5)
	ID9010	(6)
macro	DATE SUPPLIER AMOUNT	(7)
DATE		
dictionary-code	A	
attribute-count	1	
output-conversion	D2/	(8)

input-conversion	D	(9)
	CU	(10)
SUPPLIER		
dictionary-code	A	
attribute-count	2	
correlative	TSUPPLIER;C;;1	(11)
input-conversion	ISUPPLIER;A1	(12)
AMOUNT		
dictionary-code	A	
attribute-count	3	
output-conversion	MD2	(13)
input-conversion	MD2	(14)

Figure 10.11: The DICT sections of the files

In Figure 10.11, the bracketed references to the left of the important fields denote:

- 1) 19876A1 is established by the CREATE-INDEX command which we use to set up the index on attribute 1 (the name) of the SUPPLIER file.
- 2) The macro indicates the default list of datanames:

NAME ADDRESS PO.NUMBER

which will be used when we issue commands such as:

UPDATE SUPPLIER
LIST SUPPLIER

- 3) This input-conversion indicates that there is an index based upon the item-id of the PURCHASE.ORDER. When this field is updated, only an order-number in this index may be entered into this field.
- 4) BSUPPLIER;2;3 indicates that there is a bridge between attribute 2 of this (the PURCHASE.ORDER file) and attribute 3 of the SUPPLIER file.
- 5) I8765A0 is established by the CREATE-INDEX command when we set up the index on attribute 0 (the item-id) of the PURCHASE.ORDER file.
- 6) ID9010 indicates that the item-ids of new items on the PURCHASE.ORDER file are to be integers and that the last one used was 9010. The next new item will have the item-id 9011 and the Update Processor will reset this code accordingly.
- 7) The macro indicates the default list of datanames:

DATE SUPPLIER AMOUNT

which will be used when we issue commands without a list of datanames, such as:

UPDATE PURCHASE.ORDER
LIST PURCHASE.ORDER

- 8) The output-conversion indicates that this is a date field which is to be displayed in the form 01/01/99.
- 9) The input-conversion indicates that this is a date field which can be entered in any of the valid date formats.
- 10) The input-conversion indicates that an existing date field may be amended character by character.
- 11) The correlative indicates that this field on the PURCHASE.ORDER file is actually the key to an item on the SUPPLIER file, and is to be shown (not as the SUPPLIER item-id, but) as attribute 1 (the name) of the SUPPLIER item.
- 12) The input-conversion indicates that there is an index based upon attribute 1 of the SUPPLIER file which may be used when entering this field.
- 13) The output conversion indicates that this field is to be displayed with two decimal places.
- 14) The input-conversion indicates that this field can be entered in forms such as 123 or 123.0 or 123.00 and is to be converted to the equivalent internal form: 12300 implying two places of decimals.

We can now use the Update Processor to maintain items on the PURCHASE.ORDER file by means of commands such as:

UPDATE PURCHASE.ORDER
to add a single item.

UPDATE PURCHASE.ORDER DATA-ENTRY
to add a sequence of items.

UPDATE PURCHASE.ORDER 9008
to maintain an existing item.

PURCHASE.ORDER	NEW ITEM
DATE	
SUPPLIER	
AMOUNT	

(a) The basic PURCHASE.ORDER display

PURCHASE.ORDER	NEW ITEM	PURCHASE.ORDER	NEW ITEM
DATE	/ <Return>	DATE	29/07/94
SUPPLIER		SUPPLIER	
AMOUNT		AMOUNT	

(b) Entering the date

PURCHASE.ORDER	NEW ITEM	PURCHASE.ORDER	NEW ITEM
DATE	29/07/94/	DATE	29/07/94
SUPPLIER	^U	SUPPLIER	LANDAU & SONS
AMOUNT		AMOUNT	

^U moves forwards through the SUPPLIER index, ^Y moves backwards through the index. When the required name is

displayed, press <Return> to select that supplier.

(c) Cruising through the SUPPLIER index

PURCHASE.ORDER NEW ITEM		PURCHASE.ORDER NEW ITEM	
DATE	29/07/94	DATE	29/07/94
SUPPLIER	LANDAU & SONS	SUPPLIER	LANDAU & SONS
AMOUNT	123<Return>	AMOUNT	123.00

(d) Entering the amount

Figure 10.12: UPDATE PURCHASE.ORDER

In each case, the fields presented for the update process will be those declared in (7). Let's imagine that we are adding a new item. Figure 10.12 (a) shows the basic screen display; (b) shows the appearance as we enter / for the date field with the result being shown on the right; (c) shows how we might select the SUPPLIER name by cruising; (d) shows the appearance as we enter 123 for the AMOUNT field. When we finally file the item (using the ^XF sequence), the item-id for the new item will be based upon the information shown in (6).

PURCHASE.ORDER	'9008' SIZE = 30
DATE	15/05/94
SUPPLIER	WATKINS LTD
AMOUNT	345.75

(a) Updating item 9008 on the PURCHASE.ORDER file

PURCHASE.ORDER	'9008' SIZE = 30	
DATE	15/05/94	
SUPPLIER	WATKINS LTD^GG	<== Zooming from here
AMOUNT	345.75	
SUPPLIER	'2000' SIZE = 50	
NAME	WATKINS LTD	
ADDRESS	123 ELLAND ROAD	
PO.NUMBER	9008	
	9010	

(b) Zooming to the SUPPLIER file

PURCHASE.ORDER	'9008' SIZE = 30	
DATE	15/05/94	
SUPPLIER	WATKINS LTD	
AMOUNT	345.75	
SUPPLIER	'2000' SIZE = 50	
NAME	WATKINS LTD	
ADDRESS	123 ELLAND ROAD	
PO.NUMBER	9008	
	9010^GG	<== Double-clutching from here
PURCHASE.ORDER	'9010' SIZE = 29	
DATE	22/05/94	
SUPPLIER	WATKINS LTD	
AMOUNT	45.50	

(c) Double-clutching to the PURCHASE.ORDER file

Figure 10.13: Zooming & double-clutching

If we are using the Update Processor to maintain an existing item, say, item 9008 on the PURCHASE.ORDER file, then the display might look like that in Figure 10.13 (a). If we move the cursor to the SUPPLIER field and press ^GG we shall **zoom** across to item 2000 (WATKINS LTD) on the SUPPLIER file, as shown in (b). Whilst we are *in* the SUPPLIER item 2000, we can make any changes to that supplier's record. We could even move the cursor to the 9010 in the PURCHASE.ORDER field in (b) and press ^GG again to zoom to the appropriate item for that item on the PURCHASE.ORDER, again we could make any changes to purchase order 9010. We could even have zoomed from SUPPLIER 2000 to purchase order item 9008, but since we are already in the midst of editing that item, the Update Processor would only allow us to look at the purchase order. This process of zooming can be continued indefinitely; the term **double-clutching** is used for zooming to more than one further file.

10.12.6 Prestored commands

When you are using the Update Processor, you may want to repeat the same sequence of commands several times. For example, you may want to make the same changes to a number of items, or make the same change at several places within a single item. The Update Processor has a facility to create and execute a **prestore command** or sequence of commands. The prestore command(s) are held in the **prestore buffer**, and a number of Update Processor commands are available for manipulating this buffer. These are illustrated in Figure 10.14.

- o **Create and modify** the contents of the prestore buffer by means of the sequence:

```
^ZL
```

The Update Processor will display:

```
P:
```

asking for the command(s) which are to be placed in the prestore buffer. The commands are entered without the <Ctrl> key, and text operands are enclosed in ' or " characters. Thus, if you wished to store the command:

```
^Axxx^Ryyy^N
```

which would replace all occurrences of xxx by yyy, you would store the sequence:

```
A'xxx'R'yyy'N
```

If there are already commands in the prestore buffer, these will be displayed and you can edit these using the normal Update Processor <Ctrl> keys.

- o **Process** the contents of the prestore buffer by means of the key:

`^P`

- **Save the contents of the prestore buffer as an item on a file by means of the sequence:**

`^ZW`

(that is, prestore Write.) and when the Update Processor asks for the **prestore-id**, enter:

`prestore.id<Return>`

specifying the name by which the prestore buffer is to be saved. Figure 10.14 (a) shows the item which is actually created.

- **Retrieve a previously saved item into the prestore buffer by means of the sequence:**

`^ZR`

(that is, prestore Read), and when the Update Processor asks for the **prestore-id**, enter:

`prestore.id<Return>`

specifying the name by which the prestore buffer was previously saved.

- **Retrieve and then execute a previously saved item by means of the sequence:**

`^ZR`

and when the Update Processor asks for the **prestore-id**, enter:

`prestore.id^P`

specifying the name by which the prestore buffer was previously saved. The saved command(s) will then be loaded into the buffer and executed.

In all cases, the **prestore-id** is entered in the form:

`{{file.name} item.id`

specifying the name of the file and the item-id. If the file name is omitted, then it is assumed that the prestore command is held on the MD.

000	prestore.id	
001	P	<= Identifies a prestore item
002	A'xxx'R'yyy'N	<= Update Processor commands

(a) Prestore item

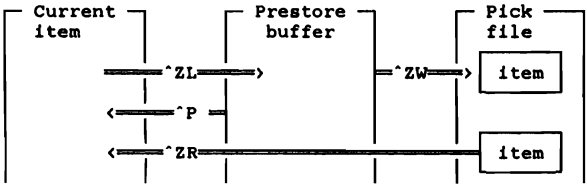




Figure 10.14: The prestore buffer

If you are creating prestore items outside the Update Processor, you should note that the format of the item is as illustrated in Figure 10.14 (a).

When an Update Processor session starts, the command:

ZN

(move the cursor to the next screen) is placed in the prestore buffer. For this reason, the command:

^P

will jump to the next page of your item. This applies until you change the contents of the prestore buffer.

10.12.7 Cutting and pasting

The Update Processor has facilities for manipulating a block of text using the text processing technique known as **cut and paste**. This is particularly useful when editing documents and Basic programs.

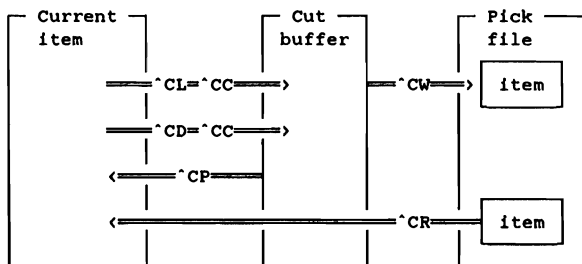


Figure 10.15: The cut buffer

There is a single **cut buffer** available for holding text which has been cut and/or is to be pasted into an item. Several <Ctrl> sequences are available for manipulating the cut buffer. They all start with ^C (for cutting), as illustrated in Figure 10.15.

- o **Cut and delete:** mark and delete a block of text, using the sequence:

^CD

to mark the start of the block, and then move the cursor to the end of the block and enter the sequence:

^CC

to mark the end of the block. The cut block is removed from the item and loaded into the cut buffer.

- o **Cut and leave:** mark a block of text and place it in the cut buffer,

using the sequence:

`^CL`

to mark the start of the block, and then move the cursor to the end of the block and enter the sequence:

`^CC`

to mark the end of the block. The item is not changed, but the marked block is loaded into the cut buffer.

- **Paste the contents of the cut buffer:** Move the cursor to the required position, and the current contents of the cut buffer are pasted into position using the sequence:

`^CP`

You should remember that the cut buffer is initialised each time you invoke the Update Processor, so it is not possible to carry text from an item in one session to an item in another (except by writing/reading the cut buffer, as described below).

- **Save the current contents of the cut buffer as an item on a file, using the sequence:**

`^CW`

(that is, Cut Write) and then, when the Update Processor asks for the **cut-id**, enter:

`cut.id <Return>`

specifying the id by which the cut section is to be saved.

- **Retrieve and paste a previously saved cut section,** the cursor is first moved to the required position.

`^CR`

(that is, Cut Read), and when the Update Processor asks for the **cut-id**, enter:

`cut.id <Return>`

specifying the id by which the cut section was previously saved. The saved cut section will then be pasted into the current item. This action is similar to the ME command of the Pick Editor.

In all cases, the cut-id is entered in the form:

`{{file.name} item.id`

specifying the name of the file and the item-id. If the file name is omitted, then it is assumed that the cut section is held on the current file.

The cut buffer is empty when the Update Processor is invoked, and the contents are lost when the UP session terminates. Thus, the buffer can be used to pass text from one item to another in the same UP session, but not between sessions.

You may find it useful to create a number of standard items – such as a skeleton document or a skeleton program – which your users may incorporate as a base or a default item when using Update Processor applications.

10.12.8 Search and replace

The Update Processor offers search and/or replace facilities. This is accomplished by a number of different sequences depending upon what you want to achieve:

- **To search only**, issue the sequence:

```
^Axxxxx<Return>
or:
^Axxxxx^M
```

to search for the specified string xxxxx.

- **To search and replace**, issue the sequence:

```
^Axxxxx^R
```

and when the system asks for a replacement string, enter the sequence:

```
yyyyy<Return>
or:
yyyyy^M
```

to search for and replace occurrences of xxxxx by yyyyy. When an occurrence is found, the system will await a further response of:

^A to ignore this occurrence and find the next.

^N to replace this and all further occurrences. **Take care with this one!** Remember that ^Z can be used to undo any accidental changes.

^R to replace this occurrence and find the next.

^X to abandon the search/replace activity.

- **To replace all occurrences**, issue the sequence:

```
^Axxxxx^R
```

and when the system asks for a replacement string, enter the sequence:

```
yyyyy^N
```

to replace all occurrences of xxxxx by yyyyy without any further intervention.

- **To repeat the last search**, issue the sequence:

```
^A
```

at any point. If, you restart the search, and there is already text in the

search buffer, you can edit this using the normal Update Processor <Ctrl> keys.

If either the search string or the replace string contains the ' or " characters, the string must be enclosed in " and ' characters respectively. Thus, you would use the command:

```
^A"'"^R'"'^N
```

to change all occurrences of ' to " in the item.

If you carry out a second search/replace, the previous strings will be retained and may be edited/cleared by the usual AP <Ctrl> keys.

Take care when using the search and replace facility: there is no *oops* facility, so it is good practice to save the item (using the ^XS sequence) before searching and replacing ... just in case.

10.12.9 Spelling check

The Update Processor has an automatic spelling checker based upon the WORDS file held on the DM account. Whenever the spelling checker is active and an unrecognised word is keyed-in, the processor will beep and allow you to enter any of:

- ^A to accept the word.
- ^A^A to accept the word and, when it is complete, add it to the WORDS file.
- ^A^A^A to disable the spelling checker.
- ^U to cruise forwards through the WORDS file and find the (alphabetically) next acceptable word.
- ^Y to cruise backwards through the WORDS file and find the previous acceptable word.
- ^ZS to switch the spelling checker on/off from within the Update Processor.

The S option on the UPDATE command will disable the spelling checker at the start of the session. The TCL commands SPELLER / SPELLER-ON / SPELLER-OFF can be used to interrogate / enable / disable the spelling checker on your port.

The WORDS file on the standard AP system is empty. Before the cruising facilities can be used, an index for the WORDS file must have been created by means of the command:

```
CREATE-INDEX WORDS A0
```

10.12.10 Hotkeys

When designing application systems around the Update Processor, the Systems Analyst can exploit the feature whereby the user can invoke a Basic subroutine by hitting a **hotkey**. A hotkey is a key sequence of the form:

$\wedge Xn$

where n is an integer 0 to 9. The names of the subroutines which are to be invoked by the hotkeys are declared in attributes 20 to 30 of the data-level identifier item for the data section of the file and/or the attribute definition for the particular dataname, as illustrated in Figure 10.16.

If the user presses a hotkey sequence when entering data via the Update Processor, the processor will look for:

- 1) The **hotkey.all** entry in the attribute definition for that data field;
- 2) The **hotkey n** entry in the attribute definition for that data field;
- 3) The **hotkey.all** entry in the data-level identifier item for that file;
- 4) The **hotkey n** entry in the data-level identifier item for that file.

and *whichever of these is found first* will be used; if there are no hotkeys defined, then no action will be taken. A typical entry for a hotkey might be:

```

20 CALL SUB.HOTKEY.ALL.STOCK
OR:
21 CALL STOCK.HOT.1
OR:
25 CALL EMERGENCY.CHECK
    
```

So, if a user is updating a particular attribute and presses the sequence:

$\wedge X1$

and the Update Processor finds the entry CALL STOCK.HOT.1 in field 21 (the **hotkey1** field) of the attribute definition for that attribute, it will call the Basic subroutine STOCK.HOT.1. Similarly, the hotkey $\wedge X5$ would call the subroutine EMERGENCY.CHECK.

The Basic subroutines which are used as hotkey routines have the familiar form, with the following observations:

- o The first statement in the subroutine will be of the form:

```
SUBROUTINE subroutine.name(parameter)
```

although the *parameter* contains only a null value and is ignored outside the subroutine.

- o No data is passed between the Basic subroutine and the data item being processed by the Update Processor.
- o The ACCESS function can be used to retrieve data about the file and the item being updated. We discuss the ACCESS function in Chapter

20	hotkey.all
21	hotkey1
22	hotkey2
23	hotkey3
24	hotkey4
25	hotkey5
26	hotkey6
27	hotkey7
28	hotkey8
29	hotkey9
30	hotkey0

Fig 10.16: Hotkey fields

14.

If several hotkey subroutines are to be called one after another, these will be specified as multivalues:

```
21 CALL STOCK.CHECK]CALL STOCK.HOT.1
```

10.13 Processing codes used with the UP

Advanced Pick uses **processing codes** in several contexts:

- As **correlatives** in attribute definitions. This specifies the way in which the data value is to be constructed from the data held within the item.
- As **conversion codes** in attribute definitions. This specifies the way in which the data is to be formatted for display.
- In the Basic ICONV and OCONV **functions** to convert data to and from the various internal/external formats which are available.
- As **input conversions** in attribute definitions and file definitions. This specifies any constraints and processing which applies to the data entered into the item via the Update Processor.

R83 Pick users will find that there are several new codes provided on Advanced Pick, as we shall see when we discuss processing codes in Chapter 14.

10.14 Summary

The Update Processor is a powerful means of creating / maintaining / deleting items. It is a dictionary-driven tool, using the attribute definitions to control the way in which the data is entered and held in the items. The Advanced Pick <Ctrl> keys are used to control the work done.

The Update Processor can also be used as a full-screen editor for creating and maintaining any items on any file, including documents, Basic programs, macros, menus.



11

The Output Processor & Runoff

The Output Processor – OP – is a text handling system provided for the production of documents, manuals, specifications, and similar printed matter on Advanced Pick. It is a standard tool within the system and is used to generate certain messages and text output from many of the standard Advanced Pick utilities.

The development and production sequence of an OP document is:

- Create, correct and maintain the document using the Update processor, and
- Print or display the document using the Output Processor.

An OP document may consist of a single piece of text, or it may be made up of a number of separate smaller clauses and paragraphs joined together into a single document as it is output. As we shall see, each document – or section of text which is to be included in another document – is created as an item on a file. The individual lines (or attributes) of the document comprise:

- The text which is to appear in the finished document, and
- Commands which control the format and the appearance of the document. These are embedded within the text of the document.

The Output Processor is provided with the sole purpose of producing printed (or displayed) material, and the commands offer facilities for:

- Formatting, justification and pagination of the text.
- Producing headings and footings on the final document.
- Numbering of chapters, sections, figures, pages, tables, and appendixes.
- Controlling the format of chapter headings and section headings.
- Producing a table of contents.
- Producing an index.
- Handling print fonts, type faces and italics.
- Controlling horizontal and vertical proportional spacing.
- Creating your own macros, invoking specific text and/or commands by including your own OP-like commands in your document.

Unlike word processing software such as *WordPerfect* and *Word*, the

Output Processor does not include its own text input or editing facility, but the Update Processor is an ideal tool for use in creating and maintaining text item(s) for the Output Processor. As we saw in the previous Chapter, the Update Processor offers facilities such as:

- o The spelling checker,
- o Cut and paste, and
- o Search and replace.

If you have used more sophisticated word processing software you will miss:

- o The function keys as a means of invoking specific word processing functions.
- o The ability to draw diagrams and illustrations.

As with the Pick system in general, the Output Processor does not handle graphics easily and there are no facilities for including artwork and diagrams within the text, beyond simple boxes and tables.

- o Facilities for performing calculations within the text.

Nevertheless, it does seem a pity that many people who, with a little effort, could learn to use the Output Processor, choose to ignore it completely in favour of other cosmetically more attractive pieces of word processing software, thereby wasting a perfectly usable – and free – software tool.

The Output Processor can also be used as a tool within Basic programs, displaying and/or printing messages, explanations, help texts and so on. Using the Output Processor in this way reduces the programming effort in formatting and outputting the text, and the fact that the text is held outside the program makes it more flexible and easier to amend the text at any time, and also means that the same text can be incorporated directly into the user documentation.

11.1 Creating and printing an OP document

A typical Output Processor item is illustrated in Figure 11.1. The raw text of a document such as this would be created as an item on a file by means of a command such as:

```
UPDATE MB.TEXT SAMPLE1
```

As you see, the item consists of text and OP commands. The final output version of this document is shown in Figure 11.2.

```
01 this is a typical Output Processor document. it is made up
   of lines of text and (optionally) Output Processor commands.
   some commonly-encountered Output Processor commands are:
02 .INDENT MARGIN 5 .NOFILL .CAP SENTENCES .LINE LENGTH 60
03
04 _ .JUSTIFY
05 _ .XJUSTIFY
06 _ .INDENT MARGIN
```

```

07  _INDEX
08  _LINE LENGTH 60
09
10  .INDENT MARGIN -5 .JUSTIFY
11  All Output Processor commands begin with a dot, and the
    commands may be specified on separate lines or they may be
    embedded within the text - provided that a space comes before
    and after the command. As the first paragraph of this
    sample illustrates, unless you include commands to indicate
    otherwise, the Output Processor will:
12  .INDENT MARGIN 5 .PARAGRAPH -2 .XJUSTIFY
13  * show a heading with the page number, item-id of the
    document, and the time and date when the output document was
    produced. The _HEADING command in the document can be used
    to overwrite this default heading. The H option on the OP
    command will suppress the page heading.
14  * break down long lines of text so that they are 70
    characters or less in length. The _LINE LENGTH command is
    used to set the length of the lines.
15  * not justify the text, that is, it will not make the right
    margin even. The _JUSTIFY command is available to make the
    left and right margins even.
16  * not capitalise the first letter of each sentence. The
    _CAP SENTENCES command is available to ensure that each
    sentence starts with a capital letter.
17  * break a hyphenated word at the hyphen if it is too long
    to fit on the line.
18
19  .INDENT MARGIN -5 .XPARAGRAPH The lines of text in your
    original document may be as long or as short as necessary.
20
21  If the document contains a _FILL command, then the long and
    short lines of the following text will be joined together and
    rearranged so as to fill up the line. text which follows a
    _NOFILL command (such as the list above) will be left
    exactly as it was typed in by the author. If a line starts
    with a space, then the Output Processor interprets this as a
    new paragraph.
22
23  Any spaces in the text are left exactly as they are. So, if
    you want two spaces after each sentence, you must type them
    in.
24
25  If you want to enter text which might be confused with an OP
    command, then you must put the override character (the _
    character) immediately in front of the full stop.

```

Figure 11.1: Output Processor item

To view the final document, you will use a command such as:

```
OP MB.TEXT SAMPLE1
```

to display the document on the screen, or:

```
OP MB.TEXT SAMPLE1 (P
```

to print a hard-copy of the document.

The final output document would look like that in Figure 11.2.

Page 1 TEXT: SAMPLE1 08:29:57 29 Jul 1995
 this is a typical Output Processor document. it is made up of
 lines of text and (optionally) Output Processor commands. some
 commonly-encountered Output Processor commands are:

.JUSTIFY
 .XJUSTIFY
 .INDENT MARGIN
 .INDEX
 .LINE LENGTH 60

All Output Processor commands begin with a dot, and the commands
 may be specified on separate lines or they may be embedded
 within the text - provided that a space comes before and after
 the command. As the first paragraph of this sample illustrates,
 unless you include commands to indicate otherwise, the Output
 Processor will:

- * Show a heading with the page number, item-id of the
 document, and the time and date when the output document
 was produced. The .HEADING command in the document can be
 used to overwrite this default heading. The H option on
 the OP command will suppress the page heading.

 Page 2 TEXT: SAMPLE1 08:29:58 29 Jul 1995

- * Break down long lines of text so that they are 70
 characters or less in length. The .LINE LENGTH command is
 used to set the length of the lines.
- * Not justify the text, that is, it will not make the right
 margin even. The .JUSTIFY command is available to make the
 left and right margins even.
- * Not capitalise the first letter of each sentence. The .CAP
 SENTENCES command is available to ensure that each sentence
 starts with a capital letter.
- * Break a hyphenated word at the hyphen if it is too long to
 fit on the line.

The lines of text in your original document may be as long or as
 short as necessary.

If the document contains a .FILL command, then the long and
 short lines of the following text will be joined together and
 rearranged so as to fill up the line. Text which follows a
 .NOFILL command (such as the list above) will be left exactly as

 Page 3 TEXT: SAMPLE1 08:29:59 29 Jul 1995

it was typed in by the author. If a line starts with a space,
 then the Output Processor interprets this as a new paragraph.

Any spaces in the text are left exactly as they are. So, if
 you want two spaces after each sentence, you must type them in.

If you want to enter text which might be confused with an OP
 command, then you must put the override character (the _
 character) immediately in front of the full stop.

Figure 11.2: Output processor document

The Output Processor is not **wysiwyg** – *what you see is what you get*. The original text and the final document are quite different in appearance. The true appearance of the document is only seen when it is finally displayed or printed on the printer. This fact, together with the requirement that the user must be familiar with the Update Processor in order to create and maintain the document, means that the Output Processor is often rejected in favour of other software.

11.2 OP command

The OP command reads a raw OP item, such as we saw earlier, and produces a displayed or a printed copy of the document. This process does not affect the form or content of the source document(s) in any way. The general format of the OP command is:

```
OP file.name {item.list} {(options)}
```

where *file.name* is the name of file holding the document(s), and *item.list* is a sequence of one or more item-ids identifying the documents which are to be output; the item list may be omitted if a select-list is available. The options are shown in Figure 11.3.

n	
n.m	
n-m	output page number <i>n</i> or pages <i>n</i> to <i>m</i> inclusive.
c	suppress any CHAIN and READ commands.
D	output the document double-spaced.
F	suppress any FOOTING commands.
H	suppress any HEADING commands.
J	suppress any HILITE commands.
K	suppress the legend when printing the document.
N	suppress the pause at the end-of-page when displaying on the screen.
P	print the document on the printer.

s	suppress bold-face, underlines, italics and fonts.
T	output the document triple-spaced.
x	analyse the usage of the words in the document and place the results into the WSYM file.

Figure 11.3: OP options

Some examples of the OP command are:

```
OP MB.TEXT SAMPLE1
OP MB.TEXT SAMPLE1 (P
OP MB.TEXT CHAP1 CHAP2 CHAP3
```

11.3 Output Processor commands

There are a great many OP commands to control the form and appearance of the final document, and these are included amongst the text of the item, either on separate lines or surrounded by spaces and embedded within the text. The commands offer a wide range of facilities for controlling the appearance of the final document; some commands set the characteristics of the document – such as line length, page depth and margin size – other commands enable and disable specific features and effects – italics on, italics off, bold-face on, bold-face off:

.CAP SENTENCES	.CS
.CENTER	.C
.ITALICS	.IT
.XITALICS	.XIT
.INDENT MARGIN 5	.IM 5
.JUSTIFY	.J
.XJUSTIFY	.XJ
.LINE LENGTH 60	.LL 60

The commands may be in upper- or lower-case and abbreviated forms are available, as shown here. A full list of the Output Processor commands is given in the Appendix. It is also possible to define your own OP commands – known as **OP macros** – and use these in your documents.

11.4 Quick OP

Don't let any fear of having to master a vast number of commands put you off using the Output Processor. If you simply want to create a document to print a piece of text, then you need only remember a few points and a small number of commands:

- o You can hold the text of your document in any item on any file.
- o The lines of the raw (input) text can be any length. The Output

Processor will break them down to the length to fit your screen or your printer, or to the line length which you specify.

- You will use the Update Processor to create the text item:

```
UPDATE file.name document.name
```

- When you have filed your document away, you will use a command such as:

```
OP file.name document.name
```

to display the document, or:

```
OP file.name document.name (P
```

to print the document, specifying the name of the file and the item where the text is held.

If you want to use more than this of the Output Processor, then a few commands will be of interest:

```
.JUSTIFY
```

make the right-hand margin even.

```
.SKIP skip a blank line.
```

```
.SKIP 2
```

skip two blank lines.

```
.INDENT MARGIN 5
```

indent the left margin five places to right.

```
.INDENT MARGIN -5
```

indent the left margin five places to the left.

```
.CENTER
```

output the following line in the centre of the page.

```
.LEFT MARGIN 8
```

set the left margin of the document 8 places from the edge of the paper.

```
.LINE LENGTH 60
```

set the page width to 60 characters.

```
.PAPER LENGTH 60
```

set the page length to 60 lines.

```
.PARAGRAPH 0
```

do not indent the first line of each paragraph. A space at the start of a line will skip a blank line and start a new paragraph.

```
.PARAGRAPH 10
```

indent the first line of each paragraph 10 places to the right.

```
.READ ffffff iiii
```

incorporate the text held in item *iiii* on file *ffff* at this point as the

document is being output.

.BEGIN PAGE

skip to a new page.

.CAP SENTENCES

change the first letter of each sentence to a capital letter.

With these commands and the following rules concerning the use of commands:

- o Each Output Processor command – or sequence of commands – may be on a separate line, or they may be embedded within the text. If in doubt, put the OP commands on a separate line;
- o When embedded in the text, there must be a single space before and after the command in addition to any spaces which are to appear in the text,

and copying the illustration shown in this Chapter, you should be able to get started with the Output Processor.

11.5 Runoff

The Runoff text processing system was provided as a standard part of the R83 Pick system and, for compatibility it is still supported on Advanced Pick systems. The Output Processor is descended from Runoff and both are provided for the production of documents, manuals, specifications, and similar printed matter. The major difference is that the OP has been extended to exploit the features of current printers and printer technology, and has provided for features such as bold-face and underlining in a more consistent manner than does Runoff.

A list of the Runoff commands is given in the Appendix. As you will see, Runoff commands are very similar to those of the Output Processor except that Runoff commands must be specified on separate lines and cannot be embedded amongst the text of the item. Otherwise, Runoff is used in the same way and for the same purposes as the Output Processor. Let us illustrate Runoff by looking at a simple document. A typical item is illustrated in Figure 11.4.

This is typical of the format of a Runoff document. It is made up of lines of text and (optionally) Runoff command lines. Some commonly-encountered Runoff commands are:

```
.INDENT MARGIN 5
.NOFILL

.NOJUSTIFY
.FILL
.INDENT MARGIN
.INDEX
.JUSTIFY
.NOFILL
.INDENT MARGIN -5
.FILL
.JUSTIFY
```

All Runoff commands begin with a dot in the first position.

Any line which does not begin with a dot is assumed to be a line of text for inclusion in the document.

Unless you include Runoff commands to indicate otherwise, Runoff will automatically:

.INDENT MARGIN 5

.PARAGRAPH -2

- * fill up the lines to the required length,
- * pad the lines with spaces so as to justify the right margin (that is, make the right margin even).
- * capitalise the first letter of each sentence.
- * indent the first line of each paragraph by five spaces.
- * produce 60-character long lines.
- * produce 23 lines per page for a document which is displayed on the screen (or 60 lines per page for printed documents), taking these details from your terminal characteristics.

.INDENT MARGIN -5

.PARAGRAPH 0

Runoff will also put two blanks between sentences.

The lines of text may be as long or as short as necessary.

If the lines of text follow a .FILL command, then the text will be rearranged so as to fill up the line.

text which follows a .NOFILL command (such as the list above) will be left exactly as it was typed in by the author.

If a line starts with a space, then Runoff interprets this as a new paragraph.

Figure 11.4: Runoff item

11.6 Creating and printing a Runoff document

This Runoff item would be created on a file by means of a command such as:

```
UPDATE MB.TEXT SAMPLE2
```

When a Runoff item is processed with a command such as:

```
RUNOFF MB.TEXT SAMPLE2
```

to display the document, or:

```
RUNOFF MB.TEXT SAMPLE2 (P
```

to print the document, then the final document would look like that in Figure 11.5.

This is typical of the format of a Runoff document. It is made up of lines of text and (optionally) Runoff command lines. Some commonly-encountered Runoff commands are:

```
.NOJUSTIFY
.FILL
.INDENT MARGIN
.INDEX
.JUSTIFY
.NOFILL
```

All Runoff commands begin with a dot in the first position. Any line which does not begin with a dot is

assumed to be a line of text for inclusion in the document.

Unless you include Runoff commands to indicate otherwise, Runoff will automatically:

- * Fill up the lines to the required length,
- * Pad the lines with spaces so as to justify the right margin (that is, make the right margin even).
- * Capitalise the first letter of each sentence.
- * Indent the first line of each paragraph by five spaces.
- * Produce 60-character long lines.
- * Produce 23 lines per page for a document which is displayed on the screen (or 60 lines per page for printed documents), taking these details from your terminal characteristics.

Runoff will also put two blanks between sentences.

The lines of text may be as long or as short as necessary. If the lines of text follow a .FILL command, then the text will be rearranged so as to fill up the line.

Text which follows a .NOFILL command (such as the list above) will be left exactly as it was typed in by the author.

If a line starts with a space, then Runoff interprets this as a new paragraph.

Figure 11.5: Runoff output document

11.7 Why Runoff?

Since every Advanced Pick system comes equipped with both the Output Processor and Runoff, and the Output Processor is more powerful, why should you bother with Runoff? If you are coming new to AP without any experience of R83 Pick or similar systems, then, if you use a text processing system at all, it will probably be the Output Processor. However, many people migrating to AP from R83 Pick and the like may have brought a considerable library of Runoff documents with them; it is for these organisations and their users that I have included this brief summary of Runoff.

11.8 RUNOFF command

The RUNOFF command is used to produce a displayed or a printed copy of a document. The general format of the command is:

```
RUNOFF file.name item.list {(options)}
```

where *file.name* is the name of file holding the document(s), and *item.list* is a sequence of one or more item-ids identifying the documents which are to be output; the item list may be omitted if a select-list is available.

The options are shown in Figure 11.6.

- n** simulate bold-face by overprinting the text *n* times.
- c** suppress the action of any .CHAIN and .READ commands in the document.
- i** output the name of the document at the head of the output.
- j** suppress the effects of the .HILITE command.
- N** suppress the pause at the end-of-page when displaying the document on the screen.
- P** send the output to the printer.
- s** suppress the bold-face and underline effects.
- u** output the entire document in UPPER-CASE.

Figure 11.6: RUNOFF options

Some examples of RUNOFF commands might be:

```
RUNOFF MB.TEXT SAMPLE1
RUNOFF MB.TEXT SAMPLE1 (P
RUNOFF MB.TEXT CHAP1 CHAP2 CHAP3
```

11.9 Quick Runoff

Don't let any fear of having to master a vast number of commands put you off using Runoff. If you simply want to create a document to print a piece of text, then you need only remember a few points and a small number of commands:

- You can hold the text of your document in any item on any file.
- The lines of the raw (input) text can be any length. Runoff will break them down to the length to fit the line length which you have specified.
- A line beginning with a space will indicate the start of a new paragraph.
- You will use one of the editors to create the text item:

```
UPDATE file.name document.name
```

- When you have filed your document away, you can use the TCL RUNOFF command:

```
RUNOFF file.name document.name
```

to display the document, or:

```
RUNOFF file.name document.name (P
```

to print the document, specifying the name of the file and the item where the text is held.

- Certain print effects are produced by means of control characters:

&	underline the following character
&^	enable underlining on all subsequent text
&\	disable underlining
@	output the following character in bold-face
@^	enable bold-face on all subsequent text
@\	disable bold-face

If you want more than this of Runoff, then a few simple commands will be of interest:

```
.NOJUSTIFY
```

stop Runoff making the right-hand margin even.

```
.SKIP 2
```

skip two blank lines.

```
.INDENT MARGIN 5
```

indent the margin five places to right.

```
.INDENT MARGIN -5
```

indent the margin five places to the left.

```
.NOFILL
```

output all subsequent text exactly as it is entered. You might use this when entering a table of information which is to appear exactly as you type it in.

```
.FILL
```

reset the .NOFILL action and cause all subsequent text to be reformatted to fit the page width.

```
.CENTER
```

output the following line in the centre of the page.

```
.LEFT MARGIN 8
```

set the left margin of the document 8 places from the edge of the paper.

```
.LINE LENGTH 79
```

set the page width to 79 characters.

```
.PARAGRAPH 10
```

indent the first line of each paragraph 10 places to the right.

.PARAGRAPH 0

do not indent the first line of each paragraph.

.PARAGRAPH -10

indent the first line of each paragraph 10 places to the left (a hanging paragraph).

.READ fffff iiii

incorporate the text which is held in item *iiii* on file *ffff* at this point as the document is being output.

.BEGIN PAGE

skip to a new page.

With these commands and the following rules concerning the use of commands:

- Each Runoff command or sequence of Runoff commands must be on a separate line and the full stop must be the first character on the line,
- Runoff commands may be in upper- or lower-case,

and copying the illustrations shown in this Chapter, you should be able to get started with Runoff.

11.10 Summary

On R83 Pick systems, the Runoff text handling system is provided as standard. This is still available on Advanced Pick, but has been superseded by the Output Processor which has many features of other word processing software and allows greater scope for using the features – fonts, subscripts, superscripts – of current printers.



12

Access : the enquiry language

The Access enquiry language is one of the most attractive features of R83 Pick and Advanced Pick for technical users and non-technical users alike. It allows users to produce their own reports and answer their own questions about the files on their database, simply by typing in English sentences.

In this Chapter, we shall look at the Access language and the grammatical structure of Access sentences.

12.1 Introducing Access

Once you have logged on to Advanced Pick, and the system has displayed the TCL prompt:

:
you can type in any Access sentence and this will be processed and the report will immediately be displayed on your screen or printed on the printer.

Page	1	STOCK	09:29:59 29 Jul 1995
STOCK.....	DESCRIPTION.....	PRICE	
8888	DESK, GREEN, ASH	56.00	
5500	TABLE, YELLOW, PINE	50.00	
2000	SETTEE, YELLOW, OAK	100.00	
1111	STOOL, GREEN, PINE	20.00	
9000	SIDEBOARD, YELLOW, ASH	130.00	
8000	SETTEE, RED, MAPLE	10.00	
1000	DESK, GREEN-BLUE, ASH	56.00	

(a) LIST STOCK DESCRIPTION PRICE

Page	1	STOCK	09:29:59 29 Jul 1995
STOCK.....	PRICE	DESCRIPTION.....	QUANTITY
8888	56.00	DESK, GREEN, ASH	64
5500	50.00	TABLE, YELLOW, PINE	15
2000	100.00	SETTEE, YELLOW, OAK	18
1111	20.00	STOOL, GREEN, PINE	63
9000	130.00	SIDEBOARD, YELLOW, ASH	99
8000	10.00	SETTEE, RED, MAPLE	6
1000	56.00	DESK, GREEN-BLUE, ASH	8

(b) LIST STOCK PRICE DESCRIPTION QUANTITY

Figure 12.1: LIST STOCK

For example, a sentence such as:

```
LIST STOCK DESCRIPTION PRICE
```

might display a report starting like that in Figure 12.1 (a), showing the item-ids of the items on the STOCK file, together with the DESCRIPTION and the PRICE of each item. Words such as DESCRIPTION and PRICE are **datanames** which identify parts of the data on the STOCK file. You – or your Systems Analyst – will have set up these datanames on the **dictionary** of the STOCK file.

Each file has its own dictionary and its own datanames. For example, the dictionary for the STOCK file might hold datanames such as DESCRIPTION, PRICE, QUANTITY, MINIMUM-LEVEL, VALUE, LOCATION, and SUPPLIER, words which refer to the fields and data held in the STOCK items; whilst the dictionary for the STAFF file might have datanames such as NAME, SURNAME, INITIALS, ADDRESS, TELEPHONE-NUMBER, START-DATE, DATE-OF-BIRTH, and so on, words which refer to the fields and data held on the STAFF file.

If we had typed the sentence:

```
LIST STOCK DESCRIPTION PRICE LPTR
```

adding the word LPTR to the sentence, then the same report would have been produced on the **printer**.

One of the beauties of Access is its **flexibility**; you can ask for any of the datanames for the file, you can ask for the datanames in any order, and you can ask for as many datanames as you want. If we had typed the sentence:

```
LIST STOCK PRICE DESCRIPTION QUANTITY
```

we might have produced the report in Figure 12.1 (b).

Each sentence is processed immediately you type it in, so if you have made a mistake, the Access processor will tell you and reject the sentence without any fuss.

12.2 Format of Access sentences

Just as a sentence in the English language or the French language is made up of nouns, verbs, adjectives and so on, so an Access sentence is made up of grammatical elements. The general structure of any Access sentence is:

```
verb filename {item list} {selection criteria} {sort specification}
      {output list} {modifiers} {(options)}
```

I've shown some of the elements in **braces** – curly brackets – because these are optional and can be included or omitted as required; you don't type in the curly brackets. Although I've had to spread this sentence over several lines, you will type the entire sentence as a single line, just like any other TCL command.

If you omit all the optional elements, you produce the simplest possible Access sentences:

```
LIST STOCK
COUNT STAFF
```

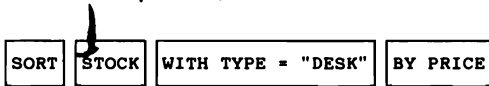
or you may have a complicated sentence, like this:

```
SORT STOCK BY TYPE BY DESCRIPTION ROLL-ON TYPE      " 'UPB'" WITH
PRICE > "50" DESCRIPTION LOCATION QUANTITY PRICE TOTAL VALUE
HEADING "STOCK REPORT : TYPE 'B' DATED 'DL'===== 'L'"
FOOTING "'LC' PAGE 'PL'" GRAND-TOTAL "FINAL TOTALS 'U'" LPTR
```

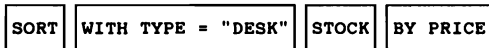
Provided that you put the **verb** at the beginning of the sentence, the rest of the elements can come in any sequence. For example, we could issue a sentence such as:



if we wanted a list of all the desks sorted in order of price; I've indicated the four elements. We could have typed this same sentence with the elements in a different sequence, such as:

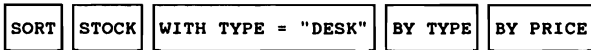


or even:

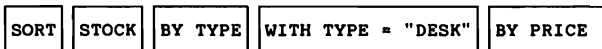


and they would all produce the same report.

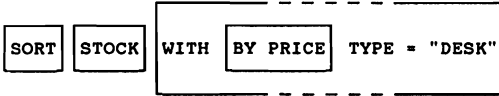
Certain elements may be repeated, like the BY elements in sentences such as:



or the equivalent:



where we sort the items first BY TYPE and then – if there are several items of the same type – BY PRICE. However, you must not split an element; so the sentence:



with the BY PRICE splitting up the WITH TYPE = "DESK" element, would be wrong and the Access processor would reject it.

12.3 Access verbs

The **verb** is the first word of an Access sentence. There are many Access verbs available, and each one has a slightly different action from the others. By far the greatest use is made of the verbs LIST and SORT.

Some of the more useful Access verbs are:

COUNT

count the number of items on a file.

HASH-TEST

produce the item statistics for a file based upon any hypothetical test modulo. We met this in Chapter 4.

ISTAT

produce the item statistics for a file. We met this in Chapter 4.

LIST

produce a report showing the items on a file and in the same order that they are held on the file.

REFORMAT

SREFORMAT

reorganise the data within the items to produce new items on another file.

SORT-ITEM

LIST-ITEM

produce a report showing the contents of the items on a file.

SORT-LABEL

LIST-LABEL

produce a report in label format.

SORT

produce a report with the items sorted into a specified sequence.

SELECT

SSELECT

create a list of the item-ids of the items on a file which can be used by another command.

STAT

produce a report showing the total and average size of items on a file.

SUM

produce a report showing the total of a data name of the items on a file.

S-DUMP

T-DUMP

dump one or more items to backing storage.

T-LOAD

restore one, several or all the items to a file from a diskette/tape produced by the T-DUMP / S-DUMP command.

12.3.1 To sort or not?

You will notice that some verbs have a sorting and a non-sorting form:

LIST-ITEM and SORT-ITEM

SELECT and SSELECT

T-DUMP and S-DUMP

and even LIST and SORT. In all cases, the sorting form is used if you specify any **sort specifications** in the sentence. Thus,

LIST STOCK BY PRICE DESCRIPTION PRICE

and:

SORT STOCK BY PRICE DESCRIPTION PRICE

are equivalent and will produce identical reports, and:

SELECT STOCK WITH MATERIAL "OAK" BY TYPE

and:

SSELECT STOCK WITH MATERIAL "OAK" BY TYPE

will produce identical results.

However, if you omit the sort specification, a sentence such as:

LIST STOCK DESCRIPTION PRICE

will present the items in the order that they are found on the file, but:

SORT STOCK DESCRIPTION PRICE

will present the items sorted into ascending order of their item-ids. This distinction applies to all the sorting/non-sorting pairs.

12.4 The file name

The next element of an Access sentence is the **file name**. This may be specified in any of the forms:

STOCK

to process the items held on the STOCK file.

CARS, DECEMBER

to process the items held on the DECEMBER data section of the CARS file.

DICT STOCK

to process the items held on the DICT section of the STOCK file.

MD

to process the items held on the MD.

You may also use the **pathnames** which we discuss in Chapter 4:

```
WAGES, ,
WAGES, PAY-RATES,
WAGES, PAY-RATES, PAY-RATES
VSALES, CARS, DECEMBER
```

12.4.1 LISTFILES

If you have forgotten the name of the file that you want to use in your Access sentences, commands such as:

LISTFILES

will display the names and other details of the files which are available to you on this account. An example of the report is shown in Figure 4.19.

LISTFILES (P

will produce the same report on the printer.

LISTFILES (C

LISTFILES (CP

will show just the names of the files on this account.

12.5 Output list

Probably the most powerful – and valuable – use of Access is in producing displayed and printed reports showing the contents of your data files. The information which appears in an Access report is:

- The item-ids,
- Any TOTALled fields,
- Any BREAK-ON fields,

together with any other datanames which you include in the sentence. The datanames for the individual pieces of information which are to appear in the report make up what is known as the **output list**. Thus, the sentence:

```
LIST STOCK PRICE
```

will produce a report containing the item-ids and one column of data: the PRICE. That is, the output list comprises just the PRICE. The sentence:

```
LIST STOCK DESCRIPTION VALUE BY PRICE
```

will produce a report showing the item-ids, the DESCRIPTION and the VALUE. The output list comprises the DESCRIPTION and the VALUE. The use of the sort specification BY PRICE to specify the order in which the items appear in the report **does not** automatically include the PRICE in the output list for the report. The sentence:

```
LIST STAFF NAME TOTAL SALARY
```

will show the item-ids, the NAME and the SALARY. The use of the TOTAL keyword to add up the individual SALARY fields will include the SALARY in the output list for that report.

There is no limit to the number of datanames which you can ask for in your output list, and you may ask for any dataname several times, if you wish.

12.6 Non-columnar reports

The Access processor will attempt to produce a report with a separate column for each dataname which you included in your output list. If your output list includes so many datanames that the resultant report is too wide for the screen (or for the printer if you've asked for a printed report) then the Access processor will abandon the normal column format and display (or print) the report in a **non-columnar form going down the page** instead of going across in columns. For example, the sentence:

```
LIST STOCK DESCRIPTION LOCATION SUPPLIER.NAME SUPPLIER.ADDRESS  
SUPPLIER.TELEPHONE
```

might produce a report like the fragment in Figure 12.2.

```

STOCK : 9999
DESCRIPTION  CHAIR, BLUE, PINE
LOCATION      MN/16/69
SUPPLIER NAME  A.B.ATKINS AND CO
SUPPLIER ADDRESS  PO BOX 134 ALTRINCHAM CHESHIRE AL9 9LA
SUPPLIER TELEPHONE  061-999 2375

STOCK : 3333
DESCRIPTION  CHAIR, WHITE, LAMINATE
LOCATION      MN/19/13
SUPPLIER NAME  A.B.ATKINS AND CO
SUPPLIER ADDRESS  PO BOX 134 ALTRINCHAM CHESHIRE AL9 9LA
SUPPLIER TELEPHONE  061-999 2375

STOCK : 6666
DESCRIPTION  SIDBOARD, ORANGE, OAK
LOCATION      LS/17/33
SUPPLIER NAME  ROGER PETERSEN PLC
SUPPLIER ADDRESS  98 WALTHAM CROSS WALTHAM FOREST ESSEX
SUPPLIER TELEPHONE  081-999 4767

```

Figure 12.2: Non-columnar output

You may use the FILL keyword with an Access sentence which is too wide for the screen (or printer) and which would produce non-columnar output. In this case, the data is output with the headings and the data continuing across the page in a text-like manner, as illustrated in Figure 12.3.

```

:SORT STOCK DESCRIPTION DATE QTY FILL

STOCK : 2000      DESCRIPTION  SETTEE, YELLOW, OAK
DATE  04 JUL 94      QTY  18

STOCK : 3000      DESCRIPTION  SIDBOARD, BLUE, ASH
DATE  25 JUL 94      QTY  58

STOCK : 4200      DESCRIPTION  SETTEE, GOLD, ASH
DATE  14 JUL 94      27 JUL 94      QTY  55  10

```

Figure 12.3: FILL

12.6.1 LISTDICT

If you have forgotten which datanames you can use with a particular file, then the LISTDICT command will list the datanames which are available. The command has the form:

```
LISTDICT STOCK
```

to display the datanames on the screen, or:

```
LISTDICT STOCK (P
```

to produce the report on the printer. In either case, the report will show all the **attribute definitions** which define the datanames on the dictionary of the file. As the fragment in Figure 12.4 shows, the LISTDICT report

includes one entry for each dataname, and shows various pieces of information about the attribute definitions which are available for use with the file. The datanames shown in the first column of this report may be freely used in Access sentences which handle that file.

We'll look at the meaning of the various parts of these attribute definitions when we look at the structure of dictionary items later.

DICT STOCK CODE A/AMC TAG..... CONVERSIONS CORRELATIVES... TP MAX									
COLOR	A	1				G1 1		T	10
						G0,1			
COLOUR	A	1				G1 1		T	10
						G0,1			
DESCR	A	1	DESCRIPTIO					T	25
			N						
DESCRIPTIONA		1						T	25
MATERIAL	A	1				G2 1		T	10
TYPE	A	1				G0,1		T	10
QTY	A	2						R	3
QUANTITY	A	2						R	1
BAY	A	3				G1/1		R	10
DEPOT	A	3				G0/1		L	10
DEPOT.NAME	A	3				G0/1		T	20
						TDEPOT;C;;1			
FIRST.LOCATAON		3				UC070;1		L	10
LOCATION	A	3						L	10
SHELF	A	3				G2/1		R	10
PRICE	A	4	PRICE	MD2				R	6
MIN	A	5						R	3
MINIMUM	A	5						R	10
SUPPLIER	A	6						T	8
SUPPLIER.ADARESS		6	Supplier a					TSUPPLIER;C;;2	T 20
			ddress						
SUPPLIER.NAAE		6						TSUPPLIER;C;;1	T 20
SUPPLIER.TEAEPHONE		6	Supplier t					TSUPPLIER;C;;3	T 20
			elephone						
DATE	A	7		D2				R	9
MOVED	A	7		D2				R	9

Figure 12.4: LISTDICT STOCK

12.6.2 Temporary attribute items : A1, A2, A3

If you don't know the datanames which are available to you for a particular file, or even if there are no datanames at all for the file, then you can use the set of temporary attribute items. These have the names:

- A0 to refer to the item-id,
- A1 to refer to attribute 1 of the items,
- A2 to refer to attribute 2,

and so on, and can be used just like the regular datanames mentioned here. They may be used with any file in sentences such as:

```
SORT STOCK A1 A2 A3 A4 BY A7
SORT STOCK A1 TOTAL A2 A3 TOTAL A5
```

```
LIST MD WITH A0 = "T-]" AND WITH A1 = "P]"
```

and they may be mixed with regular datanames:

```
SORT STOCK BY A8 DESCRIPTION TOTAL PRICE
LIST STOCK WITH A1 = "[CHAIR] A1 PRICE QTY TOTAL VALUE
```

In all cases, the output produced by these temporary attribute items is left-justified and the column is expanded to fill the available space (this is achieved by the LX justification code which we'll meet later).

If the file dictionary contains actual datanames A1, A2 and so on, these will be respected and will take precedence over the temporary attribute items.

The A0, A1, A2 names continue indefinitely, although the following are of special interest:

A9998
to output a sequential number. You can think of this as a line-number on the report.

A9999
to output the length of the item in bytes.

as illustrated in Figure 12.5.

STOCK A9998.....		A9999.....
1000	1	18
2000	2	64
3000	3	66
4000	4	62
5000	5	64
6000	6	64
7000	7	62
8000	8	62
9000	9	68

Figure 12.5: Temporary attribute items

For compatibility, the names *A0, *A1 and so on up to *A13, which you may have used on your R83 Pick system, are still available on Advanced Pick.

12.6.3 USING keyword

If – and this is most unusual – you want to use the datanames of one file with the data of another, then you can use the USING keyword in a sentence such as:

```
SORT COPY.STOCK USING DICT STOCK DESCRIPTION TOTAL VALUE WITH TYPE
"DESK"
```

in which case the Access processor will look for datanames

DESCRIPTION, VALUE and TYPE on the DICT dictionary of the STOCK file, and use these to display the data items held on the COPY.STOCK file.

12.6.4 Global dictionary: file dictionary vs MD

When you use datanames in an Access sentence, the Access processor will expect these to be defined in the DICT dictionary of that file. If it cannot find definitions for the datanames there, then it will look through the MD for the definitions. If it cannot find the definitions there, then it will reject the datanames and abandon the sentence.

000	*LINES	*BYTES	*GAP	*ONE
001	A	A	A	A
002	0	9999	0	0
003	Lines	Bytes		Count
004				
005				
006				
007				
008	U201E		C ' '	C; '1'
009	R	R	L	R
010	5	5	3	5

(a)

:SORT STOCK *LINES *BYTES *GAP TOTAL *ONE				
STOCK	Lines	Bytes	Count	
1000	7	18	1	
2000	7	64	1	
3000	7	66	1	
4000	7	62	1	
5000	7	64	1	
6000	7	64	1	
7000	7	62	1	
8000	7	62	1	
9000	7	68	1	
*			9	

(b)

Figure 12.6: Some useful global definitions

This means that, if you want to create global definitions for use with any file, you can place these on the MD. Some examples of such global definitions are shown in Figure 12.6:

*LINES
 outputs the number of attributes in the item.

***BYTES**

outputs the size of the item in bytes. This is like temporary attribute item A9999 mentioned earlier.

***GAP**

outputs a blank column (three spaces in width) to give more space between the columns of the report. The heading in attribute 3 is three spaces, although you cannot see this in Figure 12.6 (a).

***ONE**

outputs a number 1. This can be added up and is useful when counting the number of items.

12.6.5 Default output list / Macro

If you issue sentences such as:

```
LIST STOCK
SORT STOCK BY TYPE
LIST STOCK WITH TYPE = "DESK"
```

which do not include an output list, then you might expect just a single column – the item-ids – to be produced. But Access will let you (or your Systems Analyst) arrange to have a standard set of datanames – known as the **default output list** or the **implicit output list** – which will be used if you don't specify an output list in your sentence. This saves typing an output list with every Access sentence. Typically, the Systems Analyst will create a default output list which includes the most important fields for each file: the name, initials and department of the STAFF file, the name and telephone number of the SUPPLIER file, the description, price, quantity and location for the STOCK file, and so on.

There are several ways of specifying a default output list:

- 1) The **output macro**: Attribute 16 of D-pointer item for the file may contain a list of datanames (separated by spaces) which will be used as the default output list.
- 2) The **macro**: Attribute 15 of D-pointer item for the file may contain a list of datanames (separated by spaces) which will be used as the default output list if no output list is specified in an Access sentence and the output-macro (in attribute 16) is null.
- 3) **Numeric datanames**: If there are datanames 1, 2, 3, 4 and so on held on the DICT of the file, these will be used if no output macro and no macro can be found.
- 4) A **predefined phrase**: a phrase with the name @file.name or @ may be created on the file dictionary, as we discuss later.

The Access processor uses whichever of these it finds first. If all are absent, then no output list will be used and you will just get a list of item-ids.

If your Access sentence includes an output list:

```
LIST STOCK TYPE
SORT STOCK BY TYPE BREAK-ON TYPE
SORT STOCK BY TYPE TOTAL VALUE
SORT STOCK TOTAL QUANTITY TOTAL VALUE
```

then any default output list will be ignored. If there is a default output list and you don't want it to be used, you can include the **ONLY** keyword in the sentence:

```
LIST ONLY STOCK
```

12.7 SAMPLING: selecting the items

Unless you impose any special conditions in your Access sentence, the report will normally include *all the items on the file*. This is not always desirable. For example, you may just be testing an Access sentence and only be interested in whether the sentence works or not. The **SAMPLING** keyword will enable you to restrict your report to a specific number of items on the file. For example, sentences such as:

```
LIST STOCK SAMPLING 20
SORT STOCK DESCRIPTION BY TYPE SAMPLING 20
```

will show only the first twenty items on the file, and:

```
LIST STOCK WITH TYPE = "DESK" SAMPLING 10
```

will show only the first ten items which refer to desks.

12.7.1 Selecting item-ids

In another situation, you may be interested in specific items on the file. If you know the item-ids of the items which you want, you can include these in the Access sentence and the rest of the items on the file will be ignored. For example:

```
LIST STOCK '1000' DESCRIPTION PRICE VALUE
LIST STOCK "1000" DESCRIPTION PRICE VALUE
```

either of these sentences will show only the details of product number 1000.

```
LIST STAFF 'AB/111' 'AB/117' NAME AGE ADDRESS
LIST STAFF "AB/111" "AB/117" NAME AGE ADDRESS
```

will list the details of employees AB/111 and AB/117.

Notice that the item-ids are enclosed in **apostrophes** or **quotation marks**. You may omit these if the item-ids and the datanames aren't likely to be confused:

```
LIST STOCK 1000 DESCRIPTION PRICE VALUE
LIST STAFF AB/111 AB/117 NAME AGE ADDRESS
```

If there is any conflict, a word is assumed to be a **dataname**.

If you want a **range** of item-ids, then it may be simpler to use forms like

these:

LIST STOCK > '1000' DESCRIPTION PRICE LOCATION
LIST STOCK > "1000" DESCRIPTION PRICE LOCATION
LIST STOCK > 1000 DESCRIPTION PRICE LOCATION
LIST STOCK GT '1000' DESCRIPTION PRICE LOCATION
any of which will list details of all products whose item-id is greater than 1000. Other items on the file will be ignored.

LIST STOCK < '2000' DESCRIPTION TOTAL VALUE
LIST STOCK LT '2000' DESCRIPTION TOTAL VALUE
to include only the items with item-ids below 2000.

LIST STOCK > '1000' AND < '2000' DESCRIPTION
LIST STOCK GT '1000' AND LT '2000' DESCRIPTION
to list details of all products with an item-id in the range 1001 to 1999 inclusive.

LIST STOCK >= '1000' AND <= '2000' DESCRIPTION
LIST STOCK GE '1000' AND LE '2000' DESCRIPTION
to list details of all products with an item-id in the range 1000 to 2000 inclusive.

LIST STAFF = "AB]" NAME
LIST STAFF EQ "AB]" NAME
to list the names of all employees whose employee-number (that is the item-id) starts with the letters AB. The square bracket means ignore anything beyond the bracket, as we shall see later.

Symbol	Meaning	Symbol	Meaning
=	Equal to	#	Not equal to
EQ	Equal to	NO	Not equal to
EQUAL	Equal to	NOT	Not equal to
IS	Equal to	NE	Not equal to
<	Less than	>	Greater than
LT	Less than	GT	Greater than
BEFORE	Less than	AFTER	Greater than
<=	Less than or equal	>=	Greater than or equal
LE	Less than or equal	GE	Greater than or equal

Figure 12.7: Comparison operators

12.7.2 Selection criteria

A more common requirement is to select only those items which contain certain data values. For example, you might want a report of all the desks, or all the products which are held at the Lincoln depot. This is done by using the:

WITH dataname = "value"

clause, or the equivalent;

IF dataname = "value"

clause, like this:

```
LIST STOCK WITH TYPE "DESK" DESCRIPTION TOTAL VALUE
LIST STOCK IF TYPE = "DESK" DESCRIPTION TOTAL VALUE
SORT STOCK WITH PRICE > "20" PRICE DESCRIPTION LOCATION
LIST STOCK WITH MATERIAL = "PINE" DESCRIPTION PRICE TOTAL VALUE
SORT STAFF WITH AGE > "60" DESCRIPTION AGE BY DATE-OF-BIRTH
```

You can combine several selections, using the word OR:

```
LIST STOCK WITH TYPE = "DESK" OR WITH MATERIAL = "OAK" DESCRIPTION
PRICE LOCATION SUPPLIER
```

to show only those items which are desks or which are made of oak, and you can combine several selections, using the word AND:

```
LIST STOCK WITH QUANTITY > "50" AND WITH TYPE = "DESK" DESCRIPTION
PRICE LOCATION SUPPLIER
```

to show only those desks of which we have more than 50 units in stock.

Note that the values which you are seeking – 50 and DESK in the last example – are enclosed in **quotation marks**. In place of the = and the > signs of our examples, you can use any of the symbols shown in Figure 12.7. If you omit the symbol, as in Figure 12.8, then the = sign is assumed.

:SORT STOCK DESCRIPTION QUANTITY WITH QUANTITY "0"		
STOCK.....	DESCRIPTION.....	QUANTITY
6000	SIDEBOARD, GOLD, OAK	0
8117	SIDEBOARD, GREY, OAK	0
		0
8763	SIDEBOARD, MAROON, ASH	10
		5
		0

Figure 12.8: WITH QUANTITY "0"

Some of these forms are worth special mention:

```
WITH QTY = "0"
WITH QTY "0"
IF QTY IS "0"
IF QTY "0"
    will show only those items for which the QUANTITY is 0, as illustrated
    in Figure 12.8. If the QUANTITY is null or 00 or -0 then these will not
    satisfy this test and will not appear in the report. If you omit the
    symbol, then = is assumed.

WITH QUANTITY NE "0"
WITH QUANTITY # "0"
WITH QUANTITY NOT "0"
WITHOUT QUANTITY "0"
```

WITHOUT QUANTITY = "0"
IFNO QUANTITY "0"

will show only those items for which the QUANTITY is not 0. This will include any which have a QUANTITY of more than 0, less than 0 or a null QUANTITY.

Incidentally, British users may have some difficulty finding the key which displays the character #. On some keyboards, it is represented by the pound sign £ and on others by the hash or sharp sign #. It's usually the character above the number 3 on the top row of the keyboard.

WITH NO QUANTITY
WITHOUT QUANTITY
IFNO QUANTITY

will show only those items which have a null QUANTITY.

Figure 12.8 demonstrates an important point: if QUANTITY is a multivalued attribute and there is at least one value which satisfies the condition, then the item will appear on the report; do you see how the quantities 10 and 5 appear alongside the quantity 0 for item 8763? If you only want those items for which every multivalue satisfies the condition, then you can use the EVERY or EACH keyword in the forms:

WITH EVERY QUANTITY = "0"
WITH EVERY QUANTITY "0"
WITH EACH QUANTITY "0"

as in Figure 12.9. This time, item 8763 does not appear.

:SORT STOCK DESCRIPTION QUANTITY		
WITH EVERY QUANTITY "0"		
STOCK.....	DESCRIPTION.....	QUANTITY
6000	SIDEBBOARD, GOLD, OAK	0
8117	SIDEBBOARD, GREY, OAK	0
		0

Figure 12.9: WITH EVERY / EACH

If you wish to see those items which have a QUANTITY of zero but suppress any non-zero multivalues (as item 8763), then you can use the form:

SORT STOCK BY-EXP QUANTITY "0" DESCRIPTION QUANTITY

as illustrated in Figure 12.10. Here, only the zero value for item 8763 appears.

STOCK....	DESCRIPTION.....	QUANTITY
6000	SIDEBBOARD, GOLD, OAK	0
8117	SIDEBBOARD, GREY, OAK	0

8117	SIDEBOARD, GREY, OAK	0
8763	SIDEBOARD, MAROON, ASH	0

Figure 12.10: Only the zero values

12.7.3 Selection criteria : partial strings

Let's imagine that you want a list of all the products on the STOCK file which are made of pine. If you were to use a sentence such as:

```
LIST STOCK WITH DESCRIPTION = "PINE"
```

then you would not find any items which satisfy this requirement, because all the STOCK items have descriptions such as:

```
CHAIR, RED, OAK
DESK, GREY, ASH
CHAIR, ORANGE, PINE
SETTEE, YELLOW, OAK
SETTEE, BROWN, OAK
```

and there is no item with a description which consists just of the four letters PINE, which is what the Access sentence asks for. The value in quotation marks must match *exactly* with the value of that dataname on the file.

For our STOCK file, we have been fortunate because the Systems Analyst has created the dataname MATERIAL which picks out the last word of the description, so you could issue the sentence:

```
LIST STOCK WITH MATERIAL = "PINE" DESCRIPTION
```

But there is another, more general way of looking for a partial string, by means of a sentence such as:

```
LIST STOCK WITH DESCRIPTION = "[PINE" DESCRIPTION
```

which displays details of any items of which the description *ends* in the letters PINE. The resultant report is illustrated in Figure 12.11.

STOCK.....	DESCRIPTION.....
5500	TABLE, YELLOW, PINE
1111	STOOL, GREEN, PINE
3500	STOOL, WHITE, PINE
1500	CHAIR, ORANGE, PINE
7500	CHAIR, BLACK, PINE
9999	CHAIR, BLUE, PINE
5555	TABLE, RED, PINE

Figure 12.11: Partial string

Here are some more examples:

LIST STOCK WITH DESCRIPTION = "S]" DESCRIPTION

will display details of any items of which the description *begins* with the letter S producing a report showing the stools and the settees on the file.

LIST STOCK WITH DESCRIPTION = "[BLUE]" DESCRIPTION

will display details of any items of which the description *contains* the letters BLUE *in any position*. This would show the blue products, the blue-green products and the green-blue products ... it would also shows any products made of, say, blueberry wood.

LIST STOCK WITH DESCRIPTION = "DESK]" AND "[PINE"

LIST STOCK WITH DESCRIPTION = "DESK]" AND WITH DESCRIPTION = "[PINE"

either of these will display details of any items whose description starts with DESK and ends with PINE.

LIST STOCK WITH DESCRIPTION = "CHAIR]" OR "DESK]"

LIST STOCK WITH DESCRIPTION = "CHAIR]" "DESK]"

either of which will display details of any items whose description starts with CHAIR or with DESK.

The brackets are used singly or in pairs, as illustrated here, and they behave rather like the * wild card character which you may have met in DOS. Access has another wild card character:

^

This represents *any single character* and behaves rather like the ? wild card character which you may have met in DOS. Thus, if I want to produce a list of all the products which have DESCRIPTIONs starting with C *something* A, then I might use the sentence:

LIST STOCK WITH DESCRIPTION = "C^A]"

or if I want all those which have a five-letter word for the type, I might use:

LIST STOCK WITH TYPE = "*****"

to produce a report which includes CHAIRs and TABLEs, but not DESKs or SETTEEs.

12.8 Sort specifications

The LIST verb produces a report showing the items in the order that they appear on the file. As we saw in Chapter 4, Advanced Pick does not hold the items in any special order, it's just the order that the items were placed when they were added to the file or changed or deleted. If we were to issue the sentence:

LIST STOCK DESCRIPTION

one day, this might show the items in one sequence, and if we issue the same sentence next day, the items might appear in a different sequence.

With most commercial reports, you will want the items to appear in some particular sequence; for example, on the STOCK report, you might want the product numbers (that is, the item-ids) to appear in numerical order, or you may want the items to be sorted into alphabetical order of description. To produce a report in which the items are sorted by item-id, you will use the:

SORT

verb, as illustrated by the fragment in Figure 12.12. This sorts the entries in the report into order of ascending item-id. The items on the disk file are unchanged by this – and any other – Access process.

```
:SORT STOCK WITH MATERIAL = "PINE" DESCRIPTION
STOCK..... DESCRIPTION.....
1111      STOOL, GREEN, PINE
1500      CHAIR, ORANGE, PINE
3500      STOOL, WHITE, PINE
5500      TABLE, YELLOW, PINE
5555      TABLE, RED, PINE
7500      CHAIR, BLACK, PINE
9999      CHAIR, BLUE, PINE
```

Figure 12.12: SORT

If you want the report to be sorted by any field other than the item-id, such as PRICE or DESCRIPTION, then you must use include a **sort specification**. This is a clause starting with the keyword BY, as illustrated in Figure 12.13.

:SORT STOCK WITH MATERIAL = "PINE"		
BY PRICE DESCRIPTION PRICE		
STOCK.....	DESCRIPTION.....	PRICE
1111	STOOL, GREEN, PINE	20.00
3500	STOOL, WHITE, PINE	20.00
1500	CHAIR, ORANGE, PINE	25.00
7500	CHAIR, BLACK, PINE	25.00
9999	CHAIR, BLUE, PINE	25.00
5500	TABLE, YELLOW, PINE	50.00
5555	TABLE, RED, PINE	50.00

:SORT STOCK WITH MATERIAL = "PINE"		
BY DESCRIPTION DESCRIPTION PRICE		
STOCK.....	DESCRIPTION.....	PRICE
7500	CHAIR, BLACK, PINE	25.00
9999	CHAIR, BLUE, PINE	25.00
1500	CHAIR, ORANGE, PINE	25.00
3500	STOOL, WHITE, PINE	20.00
1111	STOOL, GREEN, PINE	20.00
5555	TABLE, RED, PINE	50.00
5500	TABLE, YELLOW, PINE	50.00

Figure 12.13: SORT ... BY

The BY keyword outputs the data in ascending order. If you are sorting by a right-justified field (typically, with numeric fields such as the PRICE), then the prices will appear in ascending order. If you are sorting by a left-justified field (typically, with text fields such as the DESCRIPTION), then the descriptions will appear in ascending alphabetical order with certain special characters coming first, then the numbers 0 to 9, then the A's followed by the B's and so on, then the a's followed by the b's and so on.

Incidentally, even though you are sorting the report in price sequence, the PRICE won't automatically appear in the report; if you want to see the PRICE, you have to include PRICE in the output list.

If you want the highest prices first, then you can sort the report in **descending** order, like this:

```
SORT STOCK BY-DSND PRICE WITH PRICE > "20" PRICE TYPE
```

Take care that you spell BY-DSND keyword correctly.

If any of your attributes are multivalued and contain more than one piece of data, then you can use the BY-EXP keyword to **explode** the data into its separate multivalues, like this:

```
SORT STOCK BY-EXP QUANTITY DESCRIPTION QUANTITY
```

Without the BY-EXP keyword, only the **first** multivalue is used for sorting purposes, although the other multivalues appear in the order in which

they are held on the file. Compare the entries for item 8763 on the two reports in Figure 12.14.

:SORT STOCK BY QUANTITY DESCRIPTION QUANTITY		
STOCK.....	DESCRIPTION.....	QUANTITY
1000	DESK, GREEN-BLUE, ASH	8
8763	SIDEBOARD, MAROON, ASH	10
		5
		0
1200	DESK, GREY, ASH	16
		32
2000	SETTEE, YELLOW, OAK	18
4200	SETTEE, GOLD, ASH	55

:SORT STOCK BY-EXP QUANTITY DESCRIPTION QUANTITY		
STOCK.....	DESCRIPTION.....	QUANTITY
8763	SIDEBOARD, MAROON, ASH	0
8763	SIDEBOARD, MAROON, ASH	5
1000	DESK, GREEN-BLUE, ASH	8
4200	SETTEE, GOLD, ASH	10
8763	SIDEBOARD, MAROON, ASH	10
1200	DESK, GREY, ASH	16
2000	SETTEE, YELLOW, OAK	18
1200	DESK, GREY, ASH	32
4200	SETTEE, GOLD, ASH	55

Figure 12.14: BY-EXP

You may even use the BY-EXP-DSND keyword to sort the exploded multivalues into descending order:

:SORT STOCK BY-EXP-DSND QUANTITY DESCRIPTION QUANTITY

12.9 BREAK-ON and control breaks

If your Access sentence includes sort specifications such as:

BY TYPE
OR:
BY DEPARTMENT

you may find that the final report looks better if you skip a few lines between the various types or the various departments, thereby making it easier to identify the individual groups. Such a visual effect is known as a **control break** and is produced by including a clause such as:

BREAK-ON TYPE
OR:
BREAK-ON DEPARTMENT

in your Access sentence, like this:

SORT STOCK BY TYPE DESCRIPTION PRICE BREAK-ON TYPE
 OF:
 SORT STAFF NAME START-DATE BY DEPARTMENT BY NAME BREAK-ON DEPARTMENT

This is illustrated in Figure 12.15. Note that, when you use the BREAK-ON keyword, the dataname will automatically be included in the output list for your report, and in the position implied by the position of the BREAK-ON clause in the sentence.

:SORT STOCK BY TYPE BREAK-ON TYPE DESCRIPTION			
TOTAL VALUE			
STOCK	TYPE.....	DESCRIPTION.....	VALUE
1234	CHAIR	CHAIR, RED, OAK	70.00
1500	CHAIR	CHAIR, ORANGE, PINE	825.00
3333	CHAIR	CHAIR, WHITE, LAMINATE	2100.00
4500	CHAIR	CHAIR, RED, LAMINATE	450.00
7500	CHAIR	CHAIR, BLACK, PINE	2125.00
9999	CHAIR	CHAIR, BLUE, PINE	300.00
***			5870.00
1000	DESK	DESK, GREEN-BLUE, ASH	448.00
2222	DESK	DESK, BLUE, OAK	5488.00
4000	DESK	DESK, BLACK, MAPLE	3808.00
7000	DESK	DESK, WHITE, ASH	3920.00
8888	DESK	DESK, GREEN, ASH	3584.00
***			17248.00
:			
2500	TABLE	TABLE, GREEN, LAMINATE	2400.00
5500	TABLE	TABLE, YELLOW, PINE	750.00
5555	TABLE	TABLE, RED, PINE	800.00
8500	TABLE	TABLE, ORANGE, LAMINATE	2850.00
***			6800.00
**			69817.00

Figure 12.15: BREAK-ON

If you include the TOTAL keyword to produce the totals for any datanames, then the subtotals will be shown at the control breaks. This is illustrated here.

As you see, a line of asterisks is displayed at each control break. You can suppress these by using a form such as:

BREAK-ON TYPE " "

which will suppress the asterisks (it actually replaces them by a space), or you could use:

BREAK-ON DEPARTMENT "=====

which will print a string of = signs (or any other message you wish) at each control break. Within the "xxxx" message after the BREAK-ON keyword, you can include any text and special options to call up other features. These include:

- to output a single apostrophe.
- B** to tie up with the B option in a HEADING or FOOTING, and insert the break-value into that HEADING or FOOTING.
- D** to suppress the control break signal when there is only one detail line.
- L** to suppress the blank line before the control break message.
- N** to set the page-number to 1 at each control break.
- P** to skip to a new page at the control break.
- R** this is used when you have several BREAK-ON clauses in the sentence and will output two or more simultaneous control breaks on the same page.
- U** to underline any TOTAL fields at the control break. We illustrate this in Figure 12.17.
- V** to output the break-value as a part of the message displayed at the control break.

Here are some examples:

```
BREAK-ON DEPARTMENT "===='P'"
to display a string of = signs and then skip to a new page at each
control break.
```

```
BREAK-ON DEPARTMENT "SUMMARY'U'"
to display the word SUMMARY and underline any subtotals produced
at each control break.
```

```
BREAK-ON TYPE "END OF 'V'S"
to display a message such as END OF CHAIRS at each control break.
```

The BREAK-ON field is always included in the output list; this may not be desirable. One way around this is to use the ROLL-ON keyword. The sentence:

```
SORT STOCK BY TYPE ROLL-ON TYPE " " DESCRIPTION TOTAL VALUE
```

will produce exactly the same output as Figure 12.15 but without the TYPE column.

You may have several control breaks:

```
SORT STAFF BY SITE BY DEPARTMENT BY NAME BREAK-ON SITE BREAK-ON
DEPARTMENT
```

Take care when you do this, because the Access processor will produce one control break whenever there is a change of SITE and another

control break whenever there is a change of DEPARTMENT: you may get more control breaks than you expect.

If you use DET-SUPP and BREAK-ON in the same sentence, the detail lines will be suppressed and – at each control break – there will appear any subtotals for any TOTALled fields, and for other fields there will appear the previous entry in that column (that is the last entry just before the control break); the item-ids will not appear. Compare the report in Figure 12.15 with that in Figure 12.16.

:SORT STOCK BY TYPE BREAK-ON TYPE DESCRIPTION TOTAL VALUE DET-SUPP		
TYPE.....	DESCRIPTION.....	VALUE
CHAIR	CHAIR, BLUE, PINE	5870.00
DESK	DESK, GREEN, ASH	17248.00
SETTEE	SETTEE, RED, MAPLE	2240.00
SIDEBOARD	SIDEBOARD, YELLOW, ASH	33540.00
STOOL	STOOL, YELLOW, LAMINATE	4119.00
TABLE	TABLE, ORANGE, LAMINATE	6800.00
		69817.00

Figure 12.16: BREAK-ON / DET-SUPP

12.10 TOTAL – totals and subtotals

Some of your datanames will represent numbers and numeric values – PRICE, QUANTITY, VALUE, AGE and so on. You can add these up as they are displayed and show the totals at the end of the report, if you include:

TOTAL VALUE
OR:
TOTAL QUANTITY

in your Access sentence, as we saw in Figure 12.15. Here are some more examples.

```
LIST STOCK DESCRIPTION DESCRIPTION PRICE TOTAL VALUE
LIST STOCK DESCRIPTION TOTAL QUANTITY PRICE TOTAL VALUE
```

The TOTAL keyword causes the totalled field to be included in the output list.

If you use the BREAK-ON keyword in your report, then the **subtotals** of any TOTALled fields will be shown at each control break. Figure 12.15 illustrates this.

12.10.1 XR processing code

This might be a good place to present the XR processing code:

XR{b}

This is used as an input conversion and produces a running subtotal of a dataname when used with the TOTAL keyword in an Access sentence. The optional *b* specification indicates the number of break-levels to which the subtotalling is to be restricted.

12.11 GRAND-TOTAL

The TOTAL keyword produces a grand total at the foot of the report beneath the QUANTITY and the VALUE columns. This grand total line will be identified by a line of asterisks at the left of the page. As the BREAK-ON keyword can specify an alternative text at the control break, so the GRAND-TOTAL keyword can be used to specify a message for the final grand totals:

```
GRAND-TOTAL "text"
```

clause in your sentence, like this:

```

SORT STOCK BY TYPE ROLL-ON TYPE " 'U'" DESCRIPTION TOTAL VALUE
GRAND-TOTAL "Final figures'U'"

```

to produce the report shown in Figure 12.17. Compare this report with those in Figures 12.15 and 12.16.

The options within the GRAND-TOTAL clause include:

- to output a single apostrophe.
- P to skip to a new page before outputting the message and the grand total.
- u to underline the grand total figures.

as in these examples:

```
GRAND-TOTAL "End of report'P'"
will skip to a new page before printing the final figures.
```

```
GRAND-TOTAL "SUMMARY'U'"
will display the word SUMMARY and underline the grand total figure.
```

```
GRAND-TOTAL "January's final figures'UP'"
will skip to a new page before printing the final figure and then underline the grand total.
```

STOCK DESCRIPTION.....		VALUE
1234	CHAIR, RED, OAK	70.00
1500	CHAIR, ORANGE, PINE	825.00
3333	CHAIR, WHITE, LAMINATE	2100.00
4500	CHAIR, RED, LAMINATE	450.00
7500	CHAIR, BLACK, PINE	2125.00
9999	CHAIR, BLUE, PINE	300.00

		5870.00
1000	DESK, GREEN-BLUE, ASH	448.00
2222	DESK, BLUE, OAK	5488.00
4000	DESK, BLACK, MAPLE	3808.00
7000	DESK, WHITE, ASH	3920.00
8888	DESK, GREEN, ASH	3584.00

		17248.00
:		
2500	TABLE, GREEN, LAMINATE	2400.00
5500	TABLE, YELLOW, PINE	750.00
5555	TABLE, RED, PINE	800.00
8500	TABLE, ORANGE, LAMINATE	2850.00

		6800.00

Final figures		69817.00

Figure 12.17: GRAND-TOTAL and subtotals

12.12 LPTR and printed reports

You will include the keyword:

LPTR

in your Access sentence if you want to print the report on the printer instead of displaying it on the screen. The word is actually an abbreviation for **line-printer**.

Here are some examples of LPTR in action:

```
LIST STOCK DESCRIPTION PRICE LPTR
LIST STOCK LPTR DESCRIPTION PRICE
```

Like the other elements which make up an Access sentence, LPTR can come anywhere in the sentence after the verb.

A shorthand way of achieving the same effect is by using the P option at the *very end of the sentence*:

```
LIST STOCK DESCRIPTION PRICE (P
```

Strictly speaking, LPTR is an **Access keyword** and may appear anywhere within an Access sentence; LPTR cannot be used in TCL commands. The (P element in just one example of a **TCL option** and can appear in any TCL command and in any Access sentence but it must appear at the very end. Thus, the TCL commands:

```
WHO (P
COPY STOCK (P
```

are acceptable, but the forms:

WHO LPTR
COPY STOCK LPTR

are not acceptable. We shall look at other Access options later.

12.12.1 TCL-SUPP

When an Access sentence produces a report *on the printer*, the sentence is normally repeated at the head of the report. If you don't want this, the action may be suppressed for an individual report by using the TCL-SUPP keyword:

LIST STOCK DESCRIPTION TCL-SUPP LPTR

The action can be disabled/enabled for all your Access reports by means of the TCL-HDR-ON and TCL-HDR-OFF commands. The TCL-HDR command will display a message indicating whether the action is in operation or not.

12.12.2 Legends and LEG-SUPP

The LEGEND facility allows the System Manager to specify a piece of text which is to appear at the foot of every Access report sent to the printer. The text of this system-wide legend is specified (in Output Processor format) in an item with the item-id LEGEND which is held on the MESSAGES file. An item containing a legend for a specific user may be created on the MESSAGES file with the item-id:

LEGEND*user.id

A simple legend might include your company name, like that illustrated in Figure 12.18.

01	
02	-----
03	Malcolm Bull / Training & Consultancy Services

Figure 12.18: Legend

As you might anticipate, if you don't want a legend, the action may be suppressed for an individual report by using the LEG-SUPP keyword:

LIST STOCK DESCRIPTION LEG-SUPP LPTR

and the action can be disabled/enabled for all your Access reports by means of the LEGEND-ON and LEGEND-OFF commands. The LEGEND command will display a message indicating whether the action is in operation or not.

12.13 HEADINGS and FOOTINGS

Access will automatically produce a **heading** at the top of every page of

your report, showing the page number, the name of the file, the date and the time, as we saw in Figure 12.1. But you may wish to specify your own heading and this can be achieved by means of the **HEADING** keyword, like this:

```

SORT STOCK DESCRIPTION PRICE HEADING "Stock Report"

```

and the words:

```

Stock Report

```

will appear at the top of each page of your report, replacing the standard heading.

If you wish, you can have your own **heading** at the top of each page and/or your own **footing** at the bottom of each page. This is illustrated by the fragment in Figure 12.19 which was produced by the sentence:

```

SORT STOCK BY PRICE DESCRIPTION PRICE
HEADING "'C'Stock Report'LC'=====L'"
FOOTING "End of page 'PN' of the report"

```

You can use **HEADINGS** and **FOOTINGS** with displayed reports and with printed reports.

You can control the text of the heading and footing by means of a set of **option codes** within the text of the heading or the footing, as we see in the following section.

Stock Report		
=====		
STOCK.....	DESCRIPTION.....	PRICE
2936	SETTEE, GREY, OAK	7.00
1234	CHAIR, RED, OAK	10.00
4545	SETTEE, BLUE, MAPLE	10.00
7777	STOOL, YELLOW, LAMINATE	17.75
:		
1111	STOOL, GREEN, PINE	20.00
7500	CHAIR, BLACK, PINE	25.00
9999	CHAIR, BLUE, PINE	25.00
End of page 1 of the report		

Figure 12.19: **HEADING / FOOTING**

12.13.1 HEADING and FOOTING options

Headings and footings are encountered in many areas of Advanced Pick: Access sentences, Basic programs, the Output Processor, and Runoff. In all these contexts, the **HEADING/FOOTING** consists of a string of text containing **option codes** enclosed in apostrophes. The following option codes are amongst those which are commonly used:

- B** to output the BREAK-ON or ROLL-ON value within the text of the heading/footing. The 'B' option must also be used in the BREAK-ON clause in the sentence.
- C** to output the text in the centre of the line.
- D** to output the date in the form: 01 Jan 1995
- L** to start a new line.
- PN** to output the page-number.
- T** to output the date and time in the form: 09:30:59 01 Jan 1995

If you wish to include an apostrophe in your heading or footing, you will represent this by two apostrophes. For example, you might use the heading:

```
HEADING "'C'*' Kathy O'Donnell's report '*LL'"
```

to output:

```
** Kathy O'Donnell's report **
```

followed by two blank lines at the top-centre of every page of the report.

If there are two adjacent codes, you will enclose them in one set of apostrophes, as these examples illustrate:

```
HEADING "'C'*' Sales Report '*LC'+++++++'L'" FOOTING  
"'LL'Produced 'T' ----- Page 'PN'"
```

A list of the heading/footing options can be found in the Appendix.

12.14 Other modifiers and connectives

There are a number of keywords – called **connectives** and **modifiers** – which will change the normal form of an Access report or modify the normal action of the Access processor. We have already met GRAND-TOTAL, BREAK-ON, BY, FILL, SAMPLING and DET-SUPP. Others which you may find useful are:

ID-SUPP

suppress the item-id which is normally shown in the first column of the report. When ID-SUPP is used, only the datanames in the output list (or the implied output list) will appear in the report.

HDR-SUPP

SUPP

either of these will suppress the normal page heading.

COL-HDR-SUPP

suppress the page heading and also the column headings.

DBL-SPC
produce double-spaced output, skipping a blank line between each item.

DUPLICATE
used with attributes for which an index is available and will allow you to select or avoid duplicate fields. The sentence:

```
LIST STOCK WITH DUPLICATE PRICE
```

will show only those items which have the same price as other items.

NI-SUPP
suppress the final message which shows the number of items listed.

NOPAGE
suppress the normal action of pausing at the foot of each page when you are displaying a report on the screen. You might use this if you are only interested in the last page of a report. This has no effect if the report is going to the printer.

As you might expect, you can put these keywords anywhere in your Access sentence:

```
LIST STOCK DESCRIPTION PRICE ID-SUPP
LIST STOCK DBL-SPC NI-SUPP
SORT STOCK BY TYPE BREAK-ON TYPE TOTAL PRICE DET-SUPP
SORT STOCK COL-HDR-SUPP ID-SUPP DBL-SPC LIST STOCK ID-SUPP DET-SUPP
ISTAT STOCK NOPAGE
```

12.14.1 SS – spreadsheet – connective

The SS spreadsheet connective is worth special mention. This produces a tabular summary of the values of one attribute between specific dates which are held in another attribute. For example, the sentence:

```
SORT STOCK SS DATED "15/07/95" "20/07/95" QTY
```

would produce a table (spreadsheet) with columns headed 15/07/95, 16/07/95, 17/07/95 and so on up to 20/07/95 showing the total of the QTY attribute for those items which have a DATED attribute on the appropriate dates, as illustrated in Figure 12.20. The output is equivalent to the control break subtotals which would have been produced by the sentence:

```
SORT STOCK BY DATED BREAK-ON DATED TOTAL QTY WITH DATED >=
"15/07/95" AND <= "20/07/95"
```

15/07/95	16/07/95	17/07/95	18/07/95	19/07/95	20/07/95	Total
55	12	89	104	82	99	441

Figure 12.20: SS connective

The general form of the SS clause is:

```
SS date.field "start.date" "end.date" total.field
```

12.14.2 Throwaway modifiers

There is another set of keywords – known as **throwaway modifiers** or **throwaway connectives** – which are provided in order to enable your users to write more natural English sentences. Thus, the sentence:

```
LIST the STOCK file WITH a PRICE of "50"
```

(where I've shown the throwaways in lower-case), is equivalent to:

```
LIST STOCK WITH PRICE "50"
```

The list of standard throwaways includes:

A	DATA	IN	OR
ANY	FILE	ITEMS	THE
ARE	FOR	OF	

and you can use these anywhere within a sentence to make it read more naturally. Throwaways are completely ignored by the Access processor. If you are likely to be creating your own attribute definitions, take care that you don't use any of these throwaways as your datanames; if you do, they will be ignored. The command:

```
LISTCONN
OR:
LISTCONN (P
```

will produce a list of the modifiers (or connectives) which are available with Access sentences.

12.15 Predefined phrases

STD1
01 H
02 BY TYPE DESCRIPTION BREAK-ON TYPE TOTAL VALUE

(a)

name
01 R
02 file.name <=== file holding phrase
03 phrase.name <=== name of phrase on that file

(b) A remote phrase

Figure 12.21: Phrases

Frequently-used components of your Access sentences may be constructed as phrases saved on the dictionary of the file and used in Access sentences relating to that file. For example, the phrase shown in Figure 12.21 (a) might be saved as an item on the dictionary of the STOCK file and then used in a sentence such as:

```
LIST STOCK STD1
```

to produce a report equivalent to the sentence:

```
LIST STOCK BY TYPE DESCRIPTION BREAK-ON TYPE
TOTAL VALUE
```

Such **predefined phrases** may contain any Access keywords and clauses, although they cannot contain the verb or the file name, and are most conveniently used to declare special output lists, sort sequences, selection criteria, headings and footings, and so on. A phrase may contain references to other phrases, although recursive references will result in an error message.

Phrases may be used in any appropriate context and may be mixed with other phrases, clauses, datanames and/or modifiers:

```
SORT STOCK > '1000" WITH PRICE > "50" STD1 LPTR
```

Typically, phrases will be held on the dictionary of a file where they are used, as in this example, or they may be held on the MD. A **remote phrase** may be set up to refer to a phrase held on another file, as shown in Figure 12.21 (b), which shows the name of the file and the item where the true phrase is held.

The item-id of the phrase may be any name which does not conflict with the standard datanames and Access keywords. A special phrase may be set up with the item-id:

```
@file.name
or:
@
```

and this will define a **default output list** to be used when no other output list has been included in the sentence, as we mentioned earlier. Thus, if you issue a sentence such as:

```
LIST STOCK
```

the Access processor will search for a phrase with the item-id **@STOCK** or **@** and append this to the sentence. If a file has *several* data sections, then a sentence such as:

```
LIST CARS,JANUARY
```

will seek a phrase called **@JANUARY** to use as the default phrase.

Care must be taken when using **numeric datanames** in a phrase. If the

first attribute of a phrase is H – as in Figure 12.21 (a) – then any numeric strings within the phrase are regarded as **item-ids**. Thus, created a phrase containing:

1 2 3 BY DESCRIPTION

would be interpreted as:

'1' '2' '3' BY DESCRIPTION

If you are using numeric datanames, the problem can be overcome by using the HZ identifier (instead of H), in which case, the numeric datanames – 1, 2 and 3 – in the phrase will be interpreted correctly. RZ is used to identify remote phrases which contain numeric datanames.

12.16 Select-lists and saved-lists

Let us imagine that you have a STAFF file and that you need to produce a series of reports concerning the employees aged 60 or over. When you issue a sentence such as:

```
LIST STAFF WITH AGE >= "60" NAME DEPARTMENT
```

the Access processor will read every item on the file to determine whether or not that employee is 60 or over. If you then need to produce another report about these same employees with a second sentence:

```
SORT STAFF WITH AGE >= "60" BY DEPARTMENT NAME BREAK-ON  
DEPARTMENT
```

then the Access processor will have to read all the items on the file once more. It will have to read every item on the file on each occasion, even though only a small percentage of the employees may be eligible for inclusion in the report. Can we simplify this?

Yes. If you are going to process the same subset of items on a file – the employees aged 60 or over – several times, then you can save yourself a lot of time by building a list of the item-ids of eligible employees, saving this list and then recalling it when you need to handle just those employees. As illustrated in Figure 12.22, you can create the list by a sequence such as:

```
SELECT STAFF WITH AGE >= "60"
```

which produces a **select-list**, that is, a list of the item-ids of all employees who satisfy the selection criteria, and then immediately save the list by issuing a command such as:

```
SAVE-LIST ELDERS
```

to save the select-list of eligible item-ids with the name ELDERS; you can use any name here. This **saved-list** will be saved on the POINTER-FILE file which is provided just to hold such lists. You may subsequently recall and use the saved-list whenever you need it. This is done by a command such as:

```
GET-LIST ELDERS
```

followed by the Access sentence to process the employees whose item-ids are in the select-list:

```
LIST STAFF NAME DEPARTMENT
```

and then again by a sequence such as:

```
GET-LIST ELDERS  
SORT STAFF BY DEPARTMENT NAME BREAK-ON DEPARTMENT
```


These two sequences have the same effect as the sentences at the top of this section, but using saved-lists makes the reports much faster. Figure 12.22 shows the screen display when this sequence of commands was used.

In this discussion I use the term **select-list** to mean a list of item-ids which you've just produced and which is *hanging around* ready to be used; the term **saved-list** refers to a list which you have saved permanently by means of the SAVE-LIST command. When you resurrect a *saved-list* by means of the GET-LIST command, it is converted into a *select-list* once more.

```
:SELECT STAFF WITH AGE >= "60"  
[404] 63 items selected out of 1074 items.  
  
:SAVE-LIST ELDERS  
list 'ELDERS' in file 'pointer-file' saved.  
  
:GET-LIST ELDERS  
[404] 63 items selected out of 1 items.  
  
:LIST STAFF NAME DEPARTMENT LPTR  
Entry #1  
  
:GET-LIST ELDERS  
[404] 63 items selected out of 1 items.  
  
:SORT STAFF BY DEPARTMENT NAME BREAK-ON DEPARTMENT LPTR  
Entry #2
```

Figure 12.22: Using select-lists and saved-lists

When you have issued the SAVE-LIST command, the list ELDERS is permanently saved – but it won't change as employees join or leave the organisation or when any employees reach their 60th birthday – so from time to time you may need to repeat the:

```
SELECT STAFF WITH AGE >= "60"  
SAVE-LIST ELDERS
```

sequence to create the saved-list again.

There is also the SSELECT verb (note the double S). This produces a list of item-ids which are sorted according to any BY specifications in the SSELECT sentence:

```
SSELECT STAFF BY DEPARTMENT WITH AGE >= "60"
```

When you generate a select-list by means of one of the:

```
SELECT  
SSELECT  
GET-LIST
```

verbs – or any of the other verbs shown below – then the next command which you give will use that select-list. Thus, if you use a sequence of commands such as:

```
SELECT STOCK WITH TYPE "DESK"
EDIT STOCK
```

you will edit only the DESK items on the STOCK file.

```
SELECT STOCK WITH TYPE "DESK"
SELECT STOCK WITH PRICE > "25.50"
SAVE-LIST DESK50
```

create and save a list of the item-ids of all the items which are desks and cost more than 25.50.

SAVE-LIST and GET-LIST have alternative forms SL and GL respectively. Other verbs associated with select-lists and saved-lists are:

```
QSELECT fffff iiii
GET-LIST fffff iiii
```

either of these commands will read item *iiii* from file *ffff* and use the individual attributes of that item to produce a select-list, in much the same way as the SELECT and SSELECT verbs create a select-list.

```
EDIT-LIST ELDER
```

invoke the Advanced Pick Editor to create a new list or to change a saved-list.

```
EL ELDERS
```

invoke the Update Processor to create a new list or to change a saved-list.

```
COPY-LIST ELDERS
```

```
CL ELDERS
```

either of these will copy a saved-list and saved it under another name, or save it as an item on an ordinary file. The options on the COPY-LIST command are much the same as those for the COPY command.

```
COPY-LIST ELDERS (T
```

display the contents of a saved-list.

```
COPY-LIST ELDERS (P
```

print the contents of a saved-list.

```
DELETE-LIST ELDERS
```

```
DL ELDERS
```

either of these will delete a saved-list. It is good practice to delete any lists when you no longer need them, thereby saving space on the system.

```
SORT-LIST ELDERS
```

sort the contents of a saved-list into ascending alphanumeric order.

Incidentally, since a saved-list may be held on any file, the SORT-LIST verb can be used in a form such as:

```

SORT-LIST MY.FILE MY.ITEM

```

to sort the contents of any item.

```

GET-LIST ELDERS
SSELECT ELDERS BY BIRTH.DATE
SAVE-LIST ELDERS

```

sort the contents of a saved-list into some other order, as indicated on the SSELECT sentence.

```

LIST-LISTS
LL

```

either of these will display the names of the lists which are currently saved on the POINTER-FILE.

```

ISELECT
ISSELECT

```

either of these will select all the items which are in the same physical group as a specific item:

```

ISELECT file.name item.id
ISSELECT file.name item.id

```

```

NSELECT

```

produce a list of items in the current select-list which *are not* present on a given file. You might use a sequence such as:

```

SELECT FILE1
NSELECT FILE2

```

produce a select-list of all those items which are present on FILE1 but not on FILE2.

FL form a single list from two saved-lists:

```

FL file1 list1 {operators} {file2} list2)

```

Lists held on two files can be combined by the set operations: **intersection** (using the = operator) to build a list of items found in both lists; **union** (+) to build a list of items found in either list; **exclusion** (-) to build a list of items found in *list1* but not in *list2*. The result is produced as a select-list. For example, the command:

```

FL POINTER-FILE LIST1 + LIST2

```

would produce a select-list containing those keys found in either or both the lists LIST1 and LIST2 which are saved on POINTER-FILE.

```

COMPARE-LIST

```

compare the contents of two saved-lists:

```

COMPARE-LIST {file1} list1 {operator} {file2} list2 {file3}
{list3}

```

reporting those items which occur in both lists (using the & operator),

not in both lists (#), in either list (=), in the concatenation of both lists (+), or in the difference of *list1* minus *list2* (-), to display the results or produce a new saved-list.

12.16.1 POINTER-FILE and saved-lists

Using the commands illustrated above, your lists will be saved on the standard POINTER-FILE file; this is shared by all users. Whenever you save a list, it is good practice to make a written note of the name of the list, when you created it, and what it is used for. You should also be sure the use LIST-LISTS and DELETE-LIST to monitor and remove any lists which are no longer needed. If you are creating and saving a list for short-term use, then you can save a little time by omitting the name of the list entirely:

```
SAVE-LIST
GET-LIST
EDIT-LIST
```

In this case, Advanced Pick will use the list saved on the standard POINTER-FILE with the name:

```
%u
```

where *u* is your user-id.

If you wish, you can save your lists on *any* file, it doesn't have to be the POINTER-FILE. It's a good idea to hold your saved-lists on your own file if you want to retain them for a long time, or if they are large and might hinder other users. To do this, you simply include the name of your file in the command which refers to the list:

```
SAVE-LIST MYLISTS ELDERS
GET-LIST MYLISTS ELDERS
```

and so on.

Although our examples use lists which are made up of item-ids, you can create lists from any attribute(s) on a file. For example:

```
SELECT STOCK SUPPLIER
```

generate a select-list consisting of the supplier codes (defined by the dataname SUPPLIER) taken from the items on the STOCK file.

```
SSELECT STOCK DESCRIPTION PRICE WITH TYPE "DESK"
```

generate a select-list consisting of the descriptions and prices of items. This is illustrated in Figure 12.23.

```
:SSELECT STOCK DESCRIPTION PRICE WITH TYPE "DESK"
[404] 10 items selected out of 38 items.
```

```
:SAVE-LIST DESKXXX
list 'DESKXXX' in file 'pointer-file' saved.
```

```
:COPY-LIST DESKXXX (T
  DESKXXX
001 DESK, GREEN-BLUE, ASH
002 5600
003 DESK, BLUE, OAK
004 5600
005 DESK, BLACK, MAPLE
006 5600
007 DESK, WHITE, ASH
008 5600
009 DESK, GREEN, ASH
010 5600
```

Figure 12.23: Select-lists with data

```
SELECT STOCK SUPPLIER WITH TYPE "DESK"
LIST SUPPLIER A1 A2
```

produce a report showing attributes 1 and 2 of all items on the SUPPLIER file which relate to those suppliers from whom we obtain desks. Complicated?

12.16.2 Primary and secondary lists

An important extension to the basic principles described here is provided by the concept of the **primary list** and the **secondary list**. If a select-list is created by any of the methods described above:

```
SELECT ...
SSELECT ...
GET-LIST ...
QSELECT ...
```

this is considered to be the *primary list*. If the S option is included in the command (or the SECONDARY keyword in an Access sentence), then the resultant list is considered to be the *secondary list*. Certain TCL verbs are able to use these two lists as the input list and the output list respectively. For example, the sequence:

```
SSELECT FILEA = "A]"
SSELECT FILEB = "Z]" (S
COPY FILEA (OD
TO: (FILEB
```

will select all items on FILEA with item-ids starting with A and use these (the primary list) as the item-list on the COPY command, and then select all items on FILEB with item-ids starting with Z and use these (the secondary list) as the item-list on the destination line. Other commands which can use the primary and secondary lists are:

```
COMPARE
COMPARE-LIST
```

The Basic READNEXT statement has been extended:

```
READNEXT key1 THEN/ELSE ...
READNEXT key2 SECONDARY THEN/ELSE ...
```

thus allowing a program to pick up keys from the secondary list and handle two select-lists at the same time.

12.17 Access options

Several options are available for use with Access sentences, as illustrated in Figure 12.24. Some of these are abbreviations for the modifiers which we met earlier in this Chapter:

- B** to suppress the number of items listed at the foot of the report. This is equivalent to the NI-SUPP modifier and the J option.
- C** to suppress the default page heading and column headings. This is equivalent to the COL-HDR-SUPP modifier.
- D** to suppress the detail lines between control breaks. This is equivalent to the DET-SUPP modifier.
- H** to suppress the default page heading. This is equivalent to the SUPP and HDR-SUPP modifiers, and will also suppress the creation of tape labels when used with T-DUMP.
- I** to suppress the output of the item-id. This is equivalent to the ID-SUPP modifier.
- K** to suppress the output of the legend message for list and sort operations. This is equivalent to the LEG-SUPP modifier.
- N** to suppress the end-of-page pause when displaying on the screen. This is equivalent to the NOPAGE modifier.
- P** to send the output to the printer. This is equivalent to the LPTR modifier.
- R** to output the data as text across the page. This is equivalent to the FILL modifier.

Figure 12.24: Access options

Like all TCL options, Access options are specified at the end of the sentence, and are preceded by a parenthesis:

```
LIST STOCK (P
```

```
LIST STOCK (P)
LIST STOCK WITH TYPE "DESK" (P
SORT STOCK BY PRICE DESCRIPTION TOTAL VALUE (CI
SORT STOCK BY PRICE DESCRIPTION TOTAL VALUE (I,C
```

The options may appear in any order, with or without separating commas and with or without the closing parenthesis.

12.18 LIST-LABEL / SORT-LABEL

1000	1111
DESK, GREEN-BLUE, ASH	STOOL, GREEN, PINE
56.00	20.00
1234	1500
CHAIR, RED, OAK	CHAIR, ORANGE, PINE
10.00	25.00
2200	2222
SETTEE, BROWN, OAK	DESK, BLUE, OAK
10.00	56.00

Figure 12.25: LIST-LABEL / SORT-LABEL

The LIST-LABEL and SORT-LABEL verbs allow you to produce a report in **label-format** instead of the more familiar column-format. For example, the sentence:

```
SORT-LABEL STOCK DESCRIPTION PRICE
```

might be used to produce output like that in Figure 12.25, each item producing a label, and the datanames in the output list – item-id, DESCRIPTION, and PRICE – each producing a line on the label. The most obvious application is for printing adhesive labels, such as might be used for names and addresses. But you can, of course, use the verbs to output to the printer on plain stationery and to the screen.

The syntax of the LIST-LABEL and SORT-LABEL sentences is identical to that of an ordinary Access sentence. However, before they can proceed, the LIST-LABEL / SORT-LABEL verbs require additional information:

- a) The number of labels across the page; [2 in our example].
- b) The number of lines per label [3]. This is the same as the number of columns which would appear on the equivalent LIST or SORT sentence.
- c) The number of blank lines between the labels [1].
- d) The width of the left-hand margin, that is, the number of spaces by which the labels are to be indented [0].
- e) The number of characters across each label [26].
- f) The number of horizontal spaces between the labels [0].

Visually, these parameters refer to the layout shown in Figure 12.26.

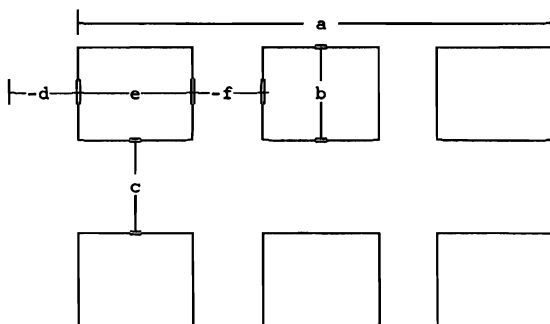


Figure 12.26: LIST-LABEL / SORT-LABEL parameters

These parameters are entered *after* the sentence has been entered and when prompted by the label-processor:

?

The values are entered at one time and are separated by commas:

2,3,1,0,26,0

If a mistake is made or the parameters are inconsistent with the page width/depth as specified by the terminal characteristics, the processor will abort with a message such as:

The range of the parameter "0" is not acceptable

Because it's easy to get the parameters wrong, the label processor is frequently invoked by a macro, a Proc or a Basic program in which the parameters have been specified correctly. This is particularly important where the facility is to be made available to end-users.

If you specify the fourth parameter (d) to indicate that you wish to indent the labels, for example:

2,3,1,10,26,0

then you will be asked to enter (again with just a sequence of ? prompts) up to ten characters of text (because we specified an indentation of 10) for each of the three lines of the label. You will specify each of the three headings one at a time when invited. These headings are only shown in front of the left-most label; labels in the second and later columns do not have the headings.

As another example, the following sequence will produce a four-column listing of the item-ids on the STOCK file:

SORT-LABEL ONLY STOCK

4.1.0.0.19.0

or we could write this as the macro or the Proc shown in Figure 12.27. Note that I have used the ONLY keyword to suppress any macro or default output list.

```
N
SORT-LABEL ONLY STOCK]4,1,0,0,19,0

PQ
HSORT-LABEL ONLY STOCK
STON
H4,1,0,0,19,0
P
```

Figure 12.27: Using SORT-LABEL

12.19 REFORMAT / SREFORMAT

The REFORMAT and SREFORMAT verbs will take the data from one file, reorganise it, and then use it to create items on another file. For example, the sentence:

REFORMAT STOCK DESCRIPTION IDATE PRICE QTY

would read the data from the items the STOCK file, rearrange the data into the order indicated by the datanames in the output list:

```
DESCRIPTION
IDATE
PRICE
QTY
```

and use it to create new items on a file specified by the user. The new items will have the same item-ids as the original items (R83 Pick users take note). If we had used a sentence such as:

REFORMAT STOCK DESCRIPTION IDATE PRICE QTY ID-SUPP

then the new items would have the DESCRIPTION (the first attribute in the output list) as their item-id. Care must be taken since any output conversions and correlatives in the attribute definitions will be applied to the new data, so a date would appear as 01 Jan 1999 instead of the internal form 11324. For this reason, it might be better to use the temporary attribute names:

```
REFORMAT STOCK A1 A7 A5 A2
```

```
1000^DESK, GREEN-BLUE, ASH^8^MN/17/81^5600^30^2000^9583
2000^SETTEE, YELLOW, OAK^18^DN/1/69^10000^30^1000^9581
3000^SIDEBOARD, BLUE, ASH^58^MN/5/56^13000^15^2000^9537
4200^SETTEE, GOLD, ASH^55]10^XN/19/35]LS/6/6^1000^30^
1000]2000]3000]4000^9559]9565
8763^SIDEBOARD, MAROON, ASH^10]5]0^LS/33/4]LN/51/7]MN/3/40^
13000^15^1000^9561]9581]9571
```

(a) The items on the STOCK file

STOCK....	DESCRIPTION.....	IDATE	PRICE	QTY
1000	DESK, GREEN-BLUE, ASH	9583	56.00	8
2000	SETTEE, YELLOW, OAK	9581	100.00	18
3000	SIDEBBOARD, BLUE, ASH	9537	130.00	58
4200	SETTEE, GOLD, ASH	9559	10.00	55
		9565		10
8763	SIDEBBOARD, MAROON, ASH	9561	130.00	10
		9581		5
		9571		0

(b) An Access report on the STOCK file

1000^DESK, GREEN-BLUE, ASH^9583^56.00^8
2000^SETTEE, YELLOW, OAK^9581^100.00^18
3000^SIDEBBOARD, BLUE, ASH^9537^130.00^58
4200^SETTEE, GOLD, ASH^9559 9565^10.00^55 10
8763^SIDEBBOARD, MAROON, ASH^9561 9581 9571^130.00^10 5 0

(c) The items on the NEW.STOCK file

Figure 12.28: Use of the REFORMAT verb

To illustrate REFORMAT, let us imagine that the STOCK file contains the items shown in Figure 12.28 (a) and that the Access sentence:

LIST STOCK DESCRIPTION IDATE PRICE QTY

produces the report shown in Figure 12.28 (b). When we issue the sentence:

REFORMAT STOCK DESCRIPTION IDATE PRICE QTY

the processor will ask us for the name of the output file:

File name?

and, if we reply with the name NEW.STOCK, the new items will be established on the NEW.STOCK file, as shown in Figure 12.28 (c).

This example illustrates some important points:

- o Any data on the source items not included in the REFORMAT sentence will be ignored when the new items are created.
- o Multivalues are transferred intact, value-marks and all.
- o Any output conversion and/or correlatives are applied to the data as it is used to build the output file.
- o REFORMAT and SREFORMAT are the only Access verbs which can actually change the data on a file. All other Access verbs perform look-only actions.

To these, we might add some further important points:

- o If an item already exists on the output file, then the new data is added as **multivalues**; R83 Pick users should note this point.
- o If you give a null response to the FILE NAME prompt, then the input file will be assumed. Care should be taken when reformatting a file back on to itself in this manner.

- If you specify TAPE as the output file, then the records will be written to backing storage.
- R83 Pick users should note the way in which the new item-ids are created. On R83 Pick, the first output value was used as the new item-id.

12.20 Access and SQL

At the time of writing, **SQL** – the structured query language – is being implemented on Advanced Pick. This will be completely compatible with Access, attribute definitions and the AP database model. SQL will be covered in future editions of this book.

12.21 Summary

Access is the database enquiry language available on the R83 Pick and the Advanced Pick operating system.

Each file has a data dictionary which holds a number of attribute definition for datanames, such as DESCRIPTION, PRICE, LOCATION, QUANTITY, which identify the attributes of the data items, or datanames such as VALUE, TYPE which are based upon the data held in the items. These datanames can be used in sentences, to display and print reports based upon the contents of the database.

The Access language allows you to control the content, format and appearance of the reports.

13

Access & attribute definitions

In the previous Chapter, we looked at the features of the Access enquiry language and the structure of Access sentences such as:

```
LIST STOCK DESCRIPTION PRICE
LIST STOCK DESCRIPTION PRICE LPTR
LIST STOCK PRICE DESCRIPTION QUANTITY TOTAL VALUE
```

In this Chapter, we shall see how Access **attribute definitions** for **datanames** such as:

```
DESCRIPTION
PRICE
QUANTITY
VALUE
```

are constructed.

13.1 Format of attribute definitions

00 item-id	
01 dictionary-code	
02 attribute-count	
03 substitute-header	
04 structure	
05	<== not used
06	<== not used
07 output-conversion	
08 correlative	
09 attribute-type	
10 column-width	

Figure 13.1: Attribute definition for Access

Each dataname – DESCRIPTION, COLOUR, PRICE, VALUE, and so on – which is to be used in an Access sentence is defined by an item on the DICT section of the file. These items are called **attribute definitions** or **attribute defining items**. The structure of an attribute definition is shown in Figure 13.1. Each field of these items plays a specific role in producing the displayed and printed output which appears when that dataname is used in a sentence.

- 0) **Item-id**: this is any alphanumeric identifier and is the **dataname** which will be used in Access sentences.
- 1) **Dictionary-code**: this is the letter A and identifies this as an attribute

definition.

- 2) **Attribute-count:** This is a number specifying which attribute in the data item is described by this definition.
- 3) **Substitute-header:** This is the text which is to be used as the column heading when this dataname is output by an Access sentence or an UPDATE command.
- 4) **Structure:** In most cases, this will be null. It is used to hold the **structure codes** which indicate whether this field is related to any others. We discuss the subject of controlling and dependent attributes later.
- 7) **Output-conversion:** this code specifies any **processing code(s)** or **conversion code(s)** which are to be used by the Access processor to change the appearance of the data immediately before it is output.
- 8) **Correlative:** this specifies any **processing code(s)** or **correlative code(s)** which describe the way in which the data is to be assembled from the data within the item. If there is no correlative, the raw data attribute (as specified by attribute 2 of the definition) will be used.
- 9) **Attribute-type:** this is an alphabetic code specifying the *justification* of the output data, and indicates the way in which the output data is to be fitted within the column of an Access report. As we see later, this will be one of L, R, T, U, I, W, WW, X.
- 10) **Column-width:** this is a number specifying the width of the column when this field appears on an Access report.

The remaining fields are concerned with the Update Processor and are not used by the Access processor.

Let's look at these parts of an attribute definition more closely.

13.1.1 The item-id / the dataname

The item-id of an attribute definition is the **dataname** by which the definition is identified in an Access sentence. This may be any string of keyboard characters, but spaces and <Ctrl> characters should not be used.

Try to choose a name that is meaningful (and easy to remember and to type).

Datanames must not conflict with the standard keywords – the modifiers and connectives – which are used in Access sentences (so you should not use datanames such as TOTAL or BREAK-ON), nor with the throwaways (such as THE or FILE) which the Access processor recognises. If you do use names such as BREAK-ON, FILE, THE or TOTAL, these will be ignored or they may be ambiguous

Some typical datanames might be:

*VALUE	ADDRESS1	MIN>QTY	T.STOCK
%TOTAL	ADDRESS2	NAME	TELEPHONE
1	AMOUNT\$	POSTCODE	TOT.VAT.%
2	CALC.PRICE	PRICE/UNIT	TOTAL.
3	DELIVER.1	STOCK1	TOTAL.STOCK
ADDRESS	DELIVER.2	LAST-PRICE	TOTAL(A)

13.1.2 Dictionary-code

Attribute 1 of an attribute definition, the **dictionary-code**, is a code which identifies this item as an attribute definition. It will usually be:

A

although you may find S and X used in some definitions: S is an older alternative to the A code; X indicates that this dataname is not to be output, and is used to suppress a particular dataname within a sequence of numeric datanames which serve as a default output list.

13.1.3 Attribute-count

Attribute 2 of an attribute definition, the **attribute-count**, is a number specifying which attribute in the data item is described by this dataname. Thus, if the attribute-count is 1, then this definition will output attribute 1 of the data items.

Certain numeric values have a special significance:

0 identifies the *item-id* of the data items. Conventionally, 0 is used if the definition is not associated with any specific attribute, as when a *correlative* is to be used to gather the data from other sources.

9998 will output an *item sequence number*: 1 for the first item on a report, 2 for the second, and so on. Do you recall the A9998 and A9999 temporary attribute items?

9999 will output the *number of bytes* in the item, that is, the item length.

13.1.4 Substitute-header

Attribute 3 of an attribute definition, the **substitute-header**, is any text which is to be used as the column heading when this dataname is used in an Access sentence.

The heading may contain any characters, including spaces, and it does not matter if this conflicts with the standard keywords (such as TOTAL), indeed, this is one useful application of the heading field.

If the substitute-header is null, the **dataname**, the *item-id* of the definition, will be used as a column heading. If the heading consists of just the backslash character:

\

then there will be no text heading, just a line of dots. If you want no heading (that is, a blank heading), you should create a substitute-header consisting of the correct number of spaces.

If the text of the heading contains the **value-mark**, this will start a new line, allowing the text to be spread over several lines of the column-heading. For example, the substitute-header:

```
MINIMUM]OF]PRODUCT
```

would produce the three-line heading:

```
MINIMUM.....
OF
PRODUCT
```

There is a relationship between the dataname (the item-id of the definition), the contents of attribute 3 (the substitute-header) and the contents of attribute 10 (the column-width). This can be summarised by saying that the width of the displayed column on the report is whichever is the greatest of:

- o The number of characters in the dataname, if the substitute-header is null;
- o The length of the substitute-header, or the longest line if it is a multiline heading;
- o The column-width.

If the width of the column heading text is less than the column width, the heading will be padded to the right with dots.

13.1.5 Structure

Attribute 4 of an attribute definition, the **structure code**, is seldom used. Its purpose is to indicate that this dataname defines a **controlling attribute** or a **dependent attribute**.

To illustrate this, let us imagine that the data items on our STAFF file hold two attributes: one (let's say this is attribute 28), holds the DIAL.CODE, such as:

```
0171
```

and the other (attribute 29) holds the PHONE.NUMBER:

```
886 9999
```

In this situation, it would *not* be meaningful to show just the PHONE.NUMBER on an Access report, since this might be:

0171 886 9999
or:
0121 886 9999
or:
0117 886 9999

and so on. We could, therefore, use the structure code, as illustrated in Figure 13.2 (a):

C;29 in the definition for DIAL.CODE indicates that the DIAL.CODE controls attribute 29 (the PHONE.NUMBER) and

D;28 in the definition for PHONE.NUMBER indicates that the PHONE.NUMBER depends upon attribute 28 (the DIAL.CODE). This means that the dependent PHONE.NUMBER cannot appear alone.

Having done this, the PHONE.NUMBER would only appear in a report where the DIAL.CODE was also requested:

LIST STAFF NAME DIAL.CODE PHONE.NUMBER

It would be ignored in a report which did *not* include the DIAL.CODE:

LIST STAFF NAME PHONE.NUMBER

If your sentence included the datanames in the wrong sequence:

LIST STAFF NAME PHONE.NUMBER DIAL.CODE

they would be shown with the controlling attribute first, as if you had typed:

LIST STAFF NAME DIAL.CODE PHONE.NUMBER

As the report in Figure 13.2 (b) shows, an asterisk denotes that PHONE.NUMBER is the dependent variable.

One dataname may control any number of dependent attributes, but a particular *dataname* may only be dependent upon one controlling attribute.

00	DIAL.CODE	PHONE.NUMBER
01	A	A
02	28	29
03	STD code	Number
04	C;29	D;28
05		
06		
07		
08		
09	R	R
10	8	15

STAFF.....	Name.....	STD code	Number.....
			*
1000	COLE	01530	532874
1076	KEMPTON	0171	886 9999
1124	RUTLAND	01482	112825
1199	HALL	0171	196 1052
1275	MENDELSKY	0171	753 7540
1359	MOSTYN	0181	687 8870
1385	STERN	0181	835 6609

(b)

Figure 13.2: Controlling and dependent attributes

13.1.6 Output-conversion

Attribute 7 of an attribute definition, the **output conversion code**, is used by the Access processor to change the format of the data as it is being output. Typical uses of conversion codes are to insert decimal points and commas into numeric values, to add a currency symbol to sums of money, to convert dates to a format such as:

```
01 Jan 1999
01/01/99
Friday
January
```

and so on.

There may be no conversion codes specified in attribute 7, or there may be several. If there are several codes, these will be separated by the value-mark and will be processed from left to right, the result of one being passed on for processing by the next.

13.1.7 Correlative

Attribute 8 of an attribute definition, the **correlative**, is used by the Access processor to assemble the data from the contents of the data items. Much of the data in an Access report is drawn straight from the data items, in which case there would be no correlative code; but other data may be derived by manipulating a piece of data or by combining two or more pieces of data – such as multiplying the quantity attribute by the price attribute – to produce a single result, the value of the stock.

There may be no processing codes specified in attribute 8, or there may be several. If there are several codes, these will be separated by the value-mark and will be processed from left to right, the result of one being passed on for processing by the next.

13.1.8 Attribute-type

Attribute 9 of an attribute definition, the **attribute-type**, indicates the way in which the output data is to be presented on the output report, and will be one of:

- L** indicating that the data is to appear **left-justified**. This is normally used with non-numeric data and text strings such as names, descriptions, addresses, and so on.

If the actual output data of type L is wider than the width of the column, then it will be broken and continued on the next line of the report.

- R** indicating that the data is to appear **right-justified**. This is normally used with numeric data such as prices, sums of money, ages, and so on, and also with dates.

If the actual output data is wider than the width of the column, then it will overflow into the column(s) to the left.

- T** indicating that the data is to appear **left-justified**. This is normally used with text strings.

If the actual output data is wider than the width of the column, then it will be broken at the nearest **space** and continued on the next line of the report. This often looks better than the effects of the L code.

- U** indicating that the data is to appear **left-justified but unformatted** within the width of the column..

If the actual output data is wider than the width of the column, then it will overflow into the column(s) to the right.

- w** indicating that, for non-columnar output, this attribute is to be processed as a piece of text by the Output Processor. You can see this code in action when you use the EPick HELP facility, where it is used to format the text.

- ww** indicating that, for non-columnar output, this attribute *and all subsequent attributes* are to be processed as a piece of text by the Output Processor.

- x** is used on conjunction with L, R or T and indicates that the column for this attribute is to be expanded to fill the width of the output report.

If several fields on a report use the X code, then each will be expanded (in proportion to their column-width) so as to fill the width of the report.

The attribute type – whether it is Right-justified or not – also affects the way in which that field is sorted when the dataname is used in a BY clause..

Note that there is no code for outputting data in the **centre** of the column. However, the XC processing code:

XC{n}

can be used to centre the output within a field *n* characters in width.

13.1.9 Column-width

Attribute 10 of an attribute definition, the **column-width**, is a number specifying the width of the print column which is to be allowed when this dataname appears in an Access report.

As we saw earlier, the actual width of the output is affected by the substitute-header and by the attribute-type, if an X is used there.

13.2 Conversion or correlative?

Attributes 7 and 8 of an attribute definition – the **conversion code** and the **correlative** – hold **processing codes** which assemble the output data (the correlative) and convert it to the required format for output (the conversion code).

Although any particular processing code may be used as a conversion code or as a correlative, the two fields perform quite distinct roles. The distinction between the actions of the two can be seen if we look at the sequence of operations as the contents of each dataname are applied to a data item:

- 1) The Access processor first picks out from the data item the attribute which is specified in field 2 of the attribute definition for that dataname.
- 2) If there is a correlative, this is applied to the data derived at (1) and/or used to incorporate data *other than* just the raw data from (1).

If there are several processing codes in the correlative field, these will be processed from left to right.

- 3) The results from (2) are then used for any sorting (BY) and/or totalling (TOTAL) which has been specified in the Access sentence.
- 4) If there is a conversion code, this is applied to the results produced at (2) to convert or reformat the data immediately prior to output.

If there are several processing codes in the conversion code field, these will be processed from left to right, the results of one being the source data for the next.

- 5) The result from (4) is displayed on the screen or printed on the printer.

This is illustrated in Figure 13.3.

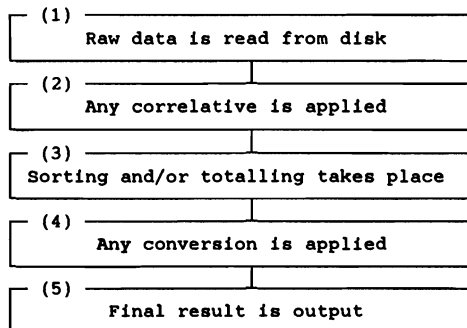


Figure 13.3: Conversions / correlatives – the sequence

13.3 Creating an attribute definition

Attribute definitions are created and maintained by means of the UD – Update Dictionary – macro:

```
UD file.name dname1 dname2
```

specifying the name of the file and a list of datanames which are to be created / changed. For example:

```
UD STOCK DESCRIPTION VALUE
```

will invoke the Update Processor and display the fields shown in Figure 10.2, allowing you to enter the information which you need. The definition is maintained by means of the regular Update Processor <Ctrl> keys. Access uses only the **dictionary-code** field (attribute 1), **attribute-count** (2), **substitute-header** (3), **structure** (4), **output-conversion** (7), **correlative** (8), **attribute-type** (9), and **column-width** (10) of the attribute definitions, the others are used by the Update Processor and may be left blank if the definition is used only in Access sentences.

13.4 Examples

In this section, we shall look at some data items and some attribute definitions, and illustrate the results when these are used in Access sentences. I have only shown that part of each output report which displays the data for items 2000 and 9000; the raw data of these items is shown in Figure 13.4 (a). The ^ character is the attribute-mark which separates the individual attributes of the data items.

2000^SETTEE, YELLOW, OAK^18^DN/1/69^10000^30^1111^9581
9000^SIDEBOARD, YELLOW, ASH^99^DN/14/70^13000^15^2222^9539

(a) Raw STOCK data

1111^JOHN WILKINSON^34 HIGH STREET^01484-123456
2222^M.A.PETERS^CARLEY IND. ESTATE^01422-987654

(b) Raw SUPPLIER data

DESCR	1	DESCRIPTION
01 A	A	A
02 1	1	1
03 Description	Product]Name	
04		
05		
06		
07		
08		
09 T	L	T
10 25	15	15

(c) Attribute definitions

:SORT STOCK DESCR 1 DESCRIPTION			
STOCK.....	Description.....	Product..... Name	DESCRIPTION....
2000	SETTEE, YELLOW, OAK	SETTEE, YELLOW, OAK	SETTEE, YELLOW, OAK
9000	SIDEBOARD, YELLOW, ASH	SIDEBOARD, YELL OW, ASH	SIDEBOARD, YELLOW, ASH

(d) Output

Figure 13.4: Substitute-header / type / column-width

13.4.1 Substitute-header / attribute-type / column-width

In Figure 13.4 (c), you see three attribute definitions all referring to attribute 1 of the data items. They differ in the substitute-header, the attribute-type and the column-width. Note how attribute-type T breaks the text at a space, whereas attribute-type L breaks it at the 15th character when the data is too wide for the column. These examples also remind

us that there may be several datanames for any particular data attribute.

13.4.2 Conversions and correlatives

QTY		MOVED	PRICE	VALUE	NAME
01	A	A	A	A	A
02	2	7	4	0	6
03			Price		Supplier]name
04					
05					
06					
07		D2	MD2	MD2	
08				A2*4	TSUPPLIER;C;;1
09	R	R	R	R	T
10	3	9	6	8	15

(a) Attribute definitions

:LIST STOCK QTY MOVED PRICE VALUE NAME					
STOCK.....	QTY	MOVED....	Price.	VALUE...	Supplier..... name
2000	18	25 MAR 94	100.00	1800.00	JOHN WILKINSON
9000	99	11 FEB 94	130.00	12870.00	M.A.PETERS

(b) Output

Figure 13.5: Conversions / correlatives

In Figure 13.5, you see four attribute definitions which use output-conversions and correlatives to process the data in Figure 13.4 (a).

- **QTY** is a simple attribute definition which takes the data from attribute 2 of the items (as indicated by line 002 of the attribute definition), and outputs this right-justified (line 009) in a column 3 characters wide (line 010). Since there is no substitute-header (line 003), the dataname (QTY) is used as the column heading.
- **MOVED** is a little more involved. This takes the data from attribute 7 of the items (line 002), then converts this to a date format (as indicated by the output conversion code D on line 007) and outputs this right-justified (line 009) in a column 8 characters wide. Since there is no substitute-header (line 003), the dataname (MOVED) is used as the column heading.
- **PRICE** takes the data from attribute 4 of the items, then inserts a decimal point to give 2 places of decimals, (as indicated by the output conversion code MD2) and outputs this right-justified in a column 6 characters wide. The substitute-header is used as the column heading.
- **VALUE** starts by taking the data from attribute 0 (the item-id) of the items, then ignores this and uses the correlative to derive a value (multiplying attribute 2 by attribute 4). This result is then passed on the output conversion code MD2 which inserts a decimal point, and then outputs this right-justified in a column 8 characters wide. Since there is no substitute-header, the dataname (VALUE) is used as the

column heading. Note that, since line 002 contains 0, the dataname would have output the item-id if this has not been overridden by the correlative.

- **NAME** takes the contents of attribute 6 of the data items, and then processes the correlative and uses the data as the key to an item on the SUPPLIER file, fetching attribute 1 (the supplier name) from that file, and outputs this as text in a column 21 characters wide. The substitute-header is used as the column heading, and the value-mark – shown as] in diagram (a) – produces a multiline heading.

These examples confirm the sequence of events discussed earlier.

13.5 Summary

The nature and appearance of the contents of an Access report is controlled by a set of attribute definitions which the Systems Analyst creates for each file. The attribute definitions are held on the DICT section of the file and specify: the dataname which is to be used in the Access sentence; which attribute of the data items which this dataname identifies; the column heading which is to appear on the report; any collection and formatting which is to be applied to the data as it is output; whether the output data is to be left- or right-justified; the width of the column on the Access report.

14

Access processing codes

Processing codes are used in several contexts within Advanced Pick:

- As **output-conversions** (attribute 7) in attribute definitions. These control the way in which the data is to be formatted for display in the final report. For example, a conversion code may be used to insert a decimal point and a currency symbol in the **VALUE** or the **PRICE** of a product. We saw this in Chapter 13.
- As **correlatives** (attribute 8) in attribute definitions. These control the way in which the data value is to be constructed from the data held within the item. For example, a correlative may be used in the attribute definition for the dataname **VALUE** showing that this is to be derived by multiplying the **PRICE** attribute by the **QUANTITY** attribute. We saw this in Chapter 13.
- As **input-conversions** (attribute 14) in attribute definitions and file definitions. These may specify any constraints and processing which applies to the data which is entered into the item via the **Update Processor**. For example, you may impose an input-conversion which sorts the multivalues of an attribute into order.
- In the Basic **ICONV** and **CONV** functions to convert data to and from the various formats which are available. For example, you might use a conversion code to convert an external date (such as 31 May 1995) to the internal format, or vice versa. We look at this in Chapter 16.

In this Chapter, we shall look at some of the available codes in detail. R83 Pick users will find that there are several new codes provided on Advanced Pick.

14.1 A : Arithmetic and logical processing

The A processing code is typically used as a correlative and allows you to perform arithmetic and other processing on the data in an item. The format of the code is:

Aexpression

where *expression* is any algebraic combination of the operands and operators described below. The expression is written in much the same way as in normal arithmetic, using brackets to indicate any precedence

in the calculation.

Here are some examples of the A code. The table in Figure 14.1 shows the results of these codes when applied to an item with the item-id XYZ which consists of the nine attributes:

11^22^33^44^55^66^77^88^1]2]3]4]5

and the current date is 9157, that is the 25th January 1993.

A2+5 will add together the contents of attributes 2 and 5 of the data item and return the result.

A7-6 will subtract the contents of attribute 6 from the contents of attribute 7 of the data item and return the result.

A(3+2)*6
will add together the contents of attributes 3 and 2 and multiply the result by the contents of attribute 6 and return the result.

A(2+5)/"2"
will add together the contents of attributes 2 and 5 of the data item and divide the result by 2. Note that the number 2 is enclosed in " or ' marks to distinguish it from attribute 2.

Code	Result
A2+5	77
A7-6	11
A(3+2)*6	3630
A(2+5)/"2"	38
A(2+5)*"100"/"2"	3850
AD-7	9080
AR(8,3)	22
A6["1","3"]	66
A0:7	XYZ77
AS(9)	15
AS(9)*7	1155
A6(MD2)	0.66
A(2*6)(MD2)	14.52
A'Y':((2*6)(MD2))	Y14.52

Figure 14.1: A processing code

All arithmetic is performed in **integers**. It is particularly important to remember this when performing division where any fractional part will be ignored, so:

A"11"/"2"
will return the value 5, and the remainder 1 (or the fractional part 0.5) will be lost.

A(2+5)*"100"/"2"
will perform the same calculation as the earlier example, but multiplying by 100 first will retain some of the decimal places. A subsequent MD2 conversion code might be used to display the number with two decimal places.

AD-will find today's date (as a number of days in the internal Pick form) and subtract from this the contents of attribute 7 of the data item. This

might be used to find the number of days since an order was placed (if the order date is held in attribute 7). Later, we'll say more about the way in which **dates** are held on data files.

AR(8,3)

will divide the contents of attribute 8 by the contents of attribute 3 and return the **remainder**.

A6["1","3"]

will return the first three characters of the contents of attribute 6.

AO:7 will **concatenate** the contents of attribute 0 (the item-id) and attribute 7 and return the result.

AS(9) will **add together all the multivalues** of attribute 9 and return the result.

AS(9)*7

will add together all the multivalues of attribute 9 and multiply the result by the contents of attribute 7.

As an alternative to specifying *attribute numbers*, you may use *datanames* in A expressions:

AN(QTY)*N(PRICE)

will evaluate the datanames for QTY and PRICE, then multiply the results together and return the product.

A(N(FLD1)+(N(FLD2))*N(PRICE))+5

will evaluate the datanames for FLD1, FLD2, and PRICE and perform the calculation shown, and then add the contents of attribute 5.

A(N(QTY)-(N(ALLOWANCE)/"1000"))*"20"

will perform the calculation using the results of evaluating the datanames for QTY and ALLOWANCE, together with the numbers 1000 and 20.

14.1.1 R for repeat

If the two attributes in the expression:

A9*4

are multivalued, then Access will perform the calculation with *corresponding* pairs of values: the first value of attribute 9 will be multiplied by the first value of attribute 4, the second value of attribute 9 by the second value of attribute 4, and so on, producing a multivalued set of products. However, if one attribute, say attribute 4, has only a single

value, then the second and subsequent values of attribute 9 will correspond with a **null** value for attribute 4 and return a result of 0. This is overcome by using the R function:

A9*4R

where the letter **R** indicates that the single value of attribute 4 is to be used **repeatedly** with each multivalued value of attribute 9.

14.1.2 Codes and codes

You may apply conversion codes to the result of an A processing code:

A6(MD2)

will apply the processing code MD2 to the contents of attribute 6.

A(2*6)(MD2)

will multiply attributes 2 and 6 together and then apply the processing code MD2 to the result.

A'Y':((2*6)(MD2))

will apply the MD2 processing to the product of attributes 2 and 6, and then prefix a Y to the result.

Figure 14.1 shows the results using the same data item as before.

14.1.3 A operands

The operands used in A expressions include:

n	the value of attribute <i>n</i> .
N(aa)	the value returned by dataname <i>aa</i> .
"n"	the numeric (or string) constant <i>n</i> .
9998	the item sequence counter, as the NI operand.
9999	the length of the item in bytes.
D	the date in the internal format.
LPV	the previous value produced.
NB	the current break-level counter.
ND	the number of details lines in this break-section.
NI	the item sequence counter, as the operand 9998.
NS	the subvalue counter.

- NV the value counter.
- R(a,b) the remainder of dividing *a* by *b*.
- S(x) the sum of multivalues in the expression *x*.
- T the time in the internal format.
- x[s,n] the string of *n* characters starting at character number *s* of string *x*.

The A code also offers logical testing which was inherited from R83 Pick, allowing you to test data and intermediate results and take one course of action if the outcome is true and another if the outcome is false. We shall not discuss these here because they have been superseded by the IF processing code on Advanced Pick.

14.1.4 A operators

The operators and operations which may be used in A expressions include:

- + addition.
- subtraction.
- * multiplication.
- / division, returning the quotient.
- : concatenation.
- (c) apply processing code *c*.

Remember that all arithmetic is done in integers.

The effects of many of the other processing codes described below can be achieved by means of the A code. This is largely for historical reasons as the older forms were superseded by the more recent A code.

14.2 B : Update Processor

The B – **bridge** – correlative is used in data-level identifiers and indicates that there is a relationship between an attribute on this file and an attribute on another file. As we see when we discuss **bridging** in Section 10.9, this is a powerful means of ensuring data integrity within the database when the fields are being handled by the Update Processor. In the example in Figure 10.5, attribute 1 of the items on the ORDER file holds the CLIENT code for the CLIENT who placed the order, and attribute 3 of the CLIENT records holds the ORDER code(s), and we use the correlative:

BCLIENT:1:3

in the data-level identifier of the ORDER file to establish such a bridge to

operate in both directions, from the ORDER file to the CLIENT and from the CLIENT file to the ORDER. Whenever an item is added to or deleted from the ORDER file, attribute 3 of the CLIENT file is changed accordingly. In this situation, the Update Processor will not allow the order-numbers on the CLIENT file to be deleted until the corresponding order on the ORDER file has been deleted.

The full form of the B code is:

```
Bfilename;a;b{:c}
```

where *filename* indicates the file (the CLIENT file in our example) to which the items on this file (the ORDER file) are associated (bridged); *a* is the attribute in the ORDER file which holds the item-ids of the CLIENT items; *b* is the attribute in the CLIENT file which forms the other end of the bridge. The parameter *c* suppresses the bridge back to this (the ORDER) file implied by the parameter *b*, and has one of the forms:

- n*:+ means **add** the value of attribute *n* of the current ORDER item to attribute *b* in the CLIENT item,
- n*:. means **subtract** the value of attribute *n* of the current ORDER item from attribute *b* in the CLIENT item,
- D* means that the current ORDER item may be **deleted** even though there is a value for attribute *a* of the item.

The following points are important when using bridges:

- o When a new order is created on the ORDER file, the order-number will be added to the corresponding item on the CLIENT file.
- so If there is no corresponding item on the CLIENT file, a new – empty – item will be created there.
- o A bridge will prevent either the ORDER record or the CLIENT record being deleted by any Advanced Pick tool (such as the Update Processor or the Editor) so long as there is a corresponding CLIENT or ORDER item on file.
- o If, say, the client-number within the ORDER item is changed, the order-number will be removed from the original CLIENT item and added to the new CLIENT item.

14.3 C : Concatenation

The C code is used to **concatenate** – join together – data attributes and/or literals to produce a single output result. It has the general form:

```
Cx{:x}
```

where the element x may be repeated as often as required in any combination of:

- a literal (enclosed in ' or " or \ characters).
- a number to indicate a source data attribute.

There may be any number of these elements separated by the semi-colon. If the element is an asterisk, then this indicates that the results on the last processing code are to be used.

To illustrate this, let us imagine that one of our data items looks like this:

```
111^222^33]44]55^333^444^555^666
```

in which attribute 1 contains the value 111, attribute 2 contains 222, and attribute 3 contains the multivalues 33, 44 and 55, and so on. The following examples show the effects of some of these codes:

`C1;"/";2`
will output **111/222** (and is identical to results produced by the `A1;"/";2` code).

`C"The answer is ";1;" units"`
will output **The answer is 111 units**

`C"£";6;".00"`
will output **£666.00**

`C"Ptas ";6;".00"`
will output **Ptas 666.00**

`C"AA";3:"BB"`
will output **AA33 44 55BB** and any value-marks will be replaced by spaces.

`C"1"`
will output the number **1** and is useful for counting items, as we see with dataname `*ONE` in Figure 12.6.

`C'YES'`
will output **YES**

A constant string may be enclosed in ' or " or \ characters.

`C""`
will output null and is useful if you want to leave a wider gap between the columns of a report, as we see with dataname `*GAP` in Figure 12.6..

C"....."
will output a line of dots.

14.4 CALL : Calling a Basic subroutine

The CALL processing code is used to call a **Basic subroutine** from an Access definition. For example, the code:

```
CALL ACCSUM
```

will call a Basic subroutine which has been catalogued with the name ACCSUM and which starts with a statement like:

```
SUBROUTINE ACCSUM(INVALUE)
```

in which a single parameter (INVALUE in this instance) is passed across containing the value derived (thus far) by the definition.

The CALL code can be used in attribute definitions used in Access reports or as an input-conversion in definitions used by the Update Processor, and opens up the way to powerful data validation and transformation. It is also used in hotkey specifications, as we see in Chapter 10.

The CALLX function is similar to CALL, but CALL is specified in an attribute definition and used only when the Access processor or the Update Processor handles that field. CALLX is held in the file definition and is used whenever *any* of the Pick processors – the EDITor, Update Processor, a Basic program, T-LOAD command, COPY command, and so on – writes items to that file.

Any processing may be done within the subroutine, including keyboard input and output. When the subroutine processing is complete (and the RETURN statement is encountered in the subroutine), the contents of the subroutine parameter (INVALUE in this case) will be passed back to the Access processor and used for any further processing by other correlatives / conversion codes, the BY connective and the TOTAL modifier, and before the result is finally output.

Within the called subroutine, details of the file and the current item can be obtained by means of the ACCESS function. This function is used in contexts such as:

```
IF ACCESS(4) = 1 THEN STORE=0
RECORD = ACCESS(3)
STORE = STORE + ACCESS(3)<2>
```

The arguments for the ACCESS function include:

- 1 returns the file-variable for the data section of the file.
- 2 returns the file-variable for the DICT section of the file. The DICT section is a useful place to store accumulators and control totals used by the subroutine.
- 3 returns the entire contents of the current item as a dynamic array; you can extract any data from this copy of the item. In certain contexts, ACCESS(3) may be changed to amend the contents of the data item.
- 4 returns the current item counter, that is, the number of items processed thus far by the Access report. In addition to its use in calculating averages and the like, this might also be used to clear and maintain running totals (for example, it might be appropriate to clear an accumulator when this has a value of 1).
- 5 returns the current attribute counter, that is, the number of the attribute which is being processed by the attribute definition.
- 10 returns the current item-id. In certain contexts, a new value may be assigned to ACCESS(10) in order to change the item-id of the current item.
- 11 returns the name of the file.

EPick gives details of the full range of facilities.

The ACCESS function can only be used within the context of subroutines invoked by the CALL and CALLX processing codes and hotkey subroutines of data-level identifiers and attribute definitions.

14.5 CU : Update Processor

When a simple data field – without any conversion codes – is being maintained via the Update Processor, the familiar <Ctrl> keys can be used to move through the field and to edit the data field. For example, a description such as:

```
CHAIR, RED, OAK
```

can easily be amended to:

```
CHAIR, GREEN, OAK
```

without having to retype CHAIR and OAK. However, if the data field is defined with an input-conversion, such as D or MT, then the entire contents of the field will be erased when the user begins to type any new data into the field. So, let us imagine that the user wishes to change an existing date:

29 Jul 1990
to:
29 Jul 1995

How can we allow the user to change just the 0 without retyping the rest of the date? The answer lies in the CU – character update – conversion code. The form of the code is simply:

CU

and will allow the <Ctrl> keys to be used to move through and edit the existing date field.

14.6 D : Date conversion

For ease of processing, dates are normally held on files as a number of days since a given starting point. For R83 Pick and Advanced Pick, the starting point (day 1) is the 1st January, 1968. Figure 14.2 shows some dates in their internal and external formats.

Day -2000 ... 10/07/62	Day 3500 31/07/77
Day -1000 ... 05/04/65	Day 4000 13/12/78
Day 0 31/12/67	Day 4500 26/04/80
Day 1 01/01/68	Day 5000 08/09/81
Day 2 02/01/68	Day 5500 21/01/83
Day 31 31/01/68	Day 6000 04/06/84
Day 32 01/02/68	Day 6500 17/10/85
Day 100 09/04/68	Day 7000 01/03/87
Day 500 14/05/69	Day 7500 13/07/88
Day 1000 26/09/70	Day 8000 25/11/89
Day 1500 08/02/72	Day 8500 09/04/91
Day 2000 22/06/73	Day 9000 21/08/92
Day 2500 04/11/74	Day 9500 03/01/94
Day 3000 18/03/76	Day 10000 ... 18/05/95

Figure 14.2: Internal and external dates

The programs which add data to the files will use the D code to convert the dates entered by the user – in forms such as 31 December 1999 or 31/12/99 – to the equivalent internal form. The D code is also used to convert internal dates to any one of a number of possible external forms. To illustrate the various D codes, let us imagine that the date attribute on file has the value:

9876

that is, **Saturday the 14th January, 1995**, then the conversion codes will return the following:

- D **14 Jan 1995**
- D2 **14 Jan 95**
- D2/ **14/01/95** (or **01/14/95** if the date is set to the standard US format).
- D4: **14:01:1995** (or **01:14:1995**).
- DD **14** (the day of the month as a number in the range 1 to 31).
- DF **January 14, 1995**
- DI This assumes that the date is held on the file in an **external form** (such as 14/1/95 or 14 Jan 1995) and will convert it to the **internal form** (9876).
- DJ **14** (the Julian date, the number of days since the beginning of the year).
- DM **1** (the month of the year as a number in the range 1 to 12).
- DMA **January**
- DQ **1** (the quarter of the year as a number in the range 1, 2, 3 or 4).
- DW **6** (the day of the week, as a number 1 for Monday to 7 for Sunday).
- DWA **Saturday**
- DY **1995**

14.6.1 DATE command

The TCL DATE command is useful when you are working with dates, as the following examples illustrate:

DATE

display today's date in the form:

July 29, 1995; Saturday Internal: 10072 Julian: 210

DATE 31/12/93

display details of a specific external date:

December 31, 1993; Friday Internal: 9497 Julian: 365

DATE 9876

display details of a specific internal date:

January 14, 1995; Saturday Internal: 9876 Julian: 14

DATE 321 (J

display details of a specific Julian date of the current year:

November 17, 1995; Friday Internal: 10183 Julian: 321

If the C option is included, the command will display a **full month's calendar**, and allow the user to browse through the calendar, using the <Ctrl> keys:

- ^D move to the previous month;
- ^F move to the following month;
- ^U move to the following year;
- ^Y move to the previous year;
- ^X abandon the process.

DATE (C

display the calendar for the current month.

DATE 1/1/49 (C

display the calendar for a specific date.

DATE 9876 (C

display a calendar for a specific internal date.

14.7 F : Arithmetic and logical processing

We have seen that the A processing code allows us to perform calculations and other operations on our data. The F – function – code offers an alternative to the A code and expresses a calculation in **reverse Polish notation** or **postfix notation**, writing the arithmetic operator *after* each pair of operands, as illustrated by the examples below. The F code is offered for compatibility with earlier versions of the Pick system, but it likely that most new work will be done using the A code. Compare these uses of the F code with the equivalent forms of the A code:

F;2;5;+

A2+5

will add together the contents of attributes 2 and 5 of the data item.

F;7;6;-

A7-6

will subtract the contents of attribute 6 from the contents of attribute 7 of the data item.

F;11;23;*

A11*23

will multiply the contents of attribute 11 by the contents of attribute 23 of the data item.

F;9;11;/

A9/11

will divide the contents of attribute 9 from the contents of attribute 11

of the data item.

F;3;2;+:6;*

A(3+2)*6

will add together the contents of attributes 3 and 2 and multiply the result by the contents of attribute 6.

F;2;5;+:C2;/

F;2;5;+: '2' ;/

A(2+5)/ '2'

will add together the contents of attributes 2 and 5 of the data item and divide the result by the number 2. Note the first form in which the constant 2 is preceded by the letter C. All arithmetic is carried out in integers and any remainder from division will be ignored.

F;D;7;-

AD-7

will find today's date (as a number of days) and subtract from this the contents of attribute 7 of the data item.

F;8;3;R

AR(8,3)

will divide the contents of attribute 8 by the contents of attribute 3 and return the remainder.

F;6;C1;C3;[]

A6['1','3']

will return the first three characters of the contents of attribute 6.

F;1;7;:

A1:7

C1:7

will concatenate the contents of attributes 1 and 7.

F;9;S

AS(9)

will add together all the multivalues of attribute 9.

F;9;S;7;*

AS(9)*7

will add together all the multivalues of attribute 9 and multiply the result by the contents of attribute 7.

EPick described the other operands and operators which are available.

Readers migrating from McDonnell Douglas systems should note the order of the operands in the F code. These are the same as the FS code on those systems and not the F code.

14.8 G : Group extraction

The G code is used to extract a part of an attribute by interpreting the value as a sequence of **groups** of data separated by a **delimiting character**. For these examples, let us assume that the data value is:

`AB/CD123/EF00/GH`

then the processing code:

`g2/1` will regard the raw data as a string made up of groups of data – AB and CD123 and EF00 and GH – separated by the character / and will then skip 2 such groups (AB and CD123) and return the next 1 group (EF00). The result would be: **EF00**

`g1/1` will return **CD123**

`g0/1` will return **AB**

`g/1` will return **AB** and is identical to the previous example.

`g1/2` will return **CD123/EF00**. Note that the delimiter(s) are included if more than one group is returned.

`g7/1` will return a null value.

`g1/999` will return **CD123/EF00/GH**, that is, everything (or the next 999 groups) after the first group.

14.9 I : File indexing / Update Processor

The I code is primarily concerned with files and their indexes, and is found in several forms:

```
I
Iacode
Ifile;acode
```

and we saw examples of some of these in Chapter 10 when we discussed cruising, zooming and double clutching.

The first form is used as an input-conversion in an attribute definition, and indicates that an index exists for this attribute. The form is simply:

```
I
```

When this attribute is processed by the Update Processor, the code allows you to **cruise** through the entries in the index by means of the Advanced Pick <Ctrl> keys.

The second form is held in the data-level identifier and is set when you issue a CREATE-INDEX command such as:

```
CREATE-INDEX STAFF A2(GO 1)
```

Alternatively, you may apply the code explicitly by setting attribute 8 of the data-level identifier to:

```
IA2(GO 1)
```

indicating that index entries are to be maintained (in this case) for the second word of attribute 2. When a CREATE-INDEX command is next issued for this file, then the index will be established and this I code will be replaced by an entry of the form:

```
I98765A2(GO 1)
```

storing the FID (98765 in this instance) of the root node of the appropriate index. Once the FID of the root node has been set by the CREATE-INDEX command, it must not be changed. Whenever an item is added, changed or deleted from this file, the index will be updated. Look at references (1) and (5) in Figure 10.10.

The third form is used as an input-conversion in an attribute definition and has a form such as:

```
ISUPPLIER:A1
```

indicating that the contents of this attribute are based upon the data which is indexed as attribute 1 of the SUPPLIER file. Look at references (3) and (12) in Figure 10.10; as reference (11) there shows, you should also have a Tfile processing code in the attribute definition for the supplier-number field of the PURCHASE.ORDER file to permit you to cruise through the SUPPLIER file. When you are required to enter a supplier-number, you can use the Advanced Pick <Ctrl> keys to pass to the index for the SUPPLIER file and browse there until the required SUPPLIER is found.

14.10 ID : Update Processor

If the Update Processor is invoked with command of the form:

```
UPDATE file.name {dataname.list} {DATA-ENTRY}
```

such as:

```
UPDATE STOCK
UPDATE STOCK DESCRIPTION PRICE
UPDATE STOCK DATA-ENTRY
UPDATE STOCK DESCRIPTION QUANTITY DATA-ENTRY
```

in which the item-id is omitted, then *a new item will be created* and when this is filed, a unique item-id will be created based upon the **current internal date** and a **system-wide sequence number**. If you wish to control the format of the new item-ids, you may use the ID code as a correlative in the data-level identifier. There are several forms of the code:

```
IDT
IDAexpression
IDn
```

The first code – IDT – will create an item-id of the form:

```
dta
```

where *d* is the internal date, *t* the internal time, and *a* is null or a sequential letter which may be appended to make the item-id unique.

The second form:

```
IDAcodes
```

for example:

```
IDA(D: '/' : T: 1[ '1' , '1' ])
```

will use the A processing code to derive the item-id from other data in the item; in this instance, building the item-id from the current date and time followed by the first letter of attribute 1 of the item. The expression can, of course, use the CALL code to invoke a Basic subroutine, thereby allowing virtually any item-id to be generated.

The third form:

```
IDn
```

will create unique numeric item-ids starting at *n*. As new items are added to the file, the system will update the value of *n* in this code to reflect the next number to be used. We used this at reference (6) in Figure 10.10. When new item-ids are generated under the control of this code and it is found that an item already exists with the generated item-id, then the Update Processor will increment the generated item-id until a unique item-id is obtained.

If an ID code is invalid or if it produces an item-id which already exists on the file, then an error is reported:

```
[73] Invalid "id" conversion specified
```

and the default item-id (today's date plus a unique identifier) will be used instead.

14.11 IF : Logical processing

The IF processing code is used in an attribute definition to test data and intermediate results and take one course of action if the outcome is true, or another if the outcome is false. The general forms of the code are:

```
IF condition THEN expression
IF condition ELSE expression
IF condition THEN expression ELSE expression
```

where *condition* is any conditional expression and *expression* is any output expression (which may include a further IF code). Some examples are:

```
IF 7<2 THEN "BELOW" ELSE "NOT BELOW"
    compare the contents of attributes 7 and 2 of the data item, and
    output the word BELOW if the value of 7 is less than the contents of
    attribute 2, otherwise the word NOT BELOW will be output.
```

```
IF 2+"0" = "0" THEN 5+7 ELSE 2
    test the contents of attribute 2 of the data item; if the value is zero or
    null, then the sum of attributes 5 and 7 will be output, otherwise the
    contents of attribute 2 will be output.
```

```
IF 2=5 THEN "OK" ELSE IF 2<5 THEN "LOW" ELSE "HIGH"
    output one of the words OK, LOW or HIGH according to the contents
    of attributes 2 and 5.
```

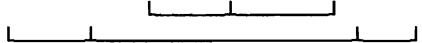
Some further examples:

```
IF 1 = "0" THEN "ZERO VALUE FOUND"
IF 1 = "0" ELSE "NON ZERO VALUE"
IF 1 > 2 THEN 1 ELSE 2
IF 2+4 > 7 THEN 2+4 ELSE 7
IF 1 AND 2 THEN 1+2 ELSE "0"
IF 1 OR 7 THEN "OK" ELSE "ERROR"
IF 1 AND 7 THEN IF 1 > 7 THEN 1 ELSE 7 ELSE "0"
IF N(VALUE) > "0" THEN "REVIEW" ELSE N(VALUE)/100
IF 1 THEN IF 2 THEN 3 ELSE 4 ELSE 5
IF N(S0) < "100" THEN N(S1) ELSE N(S2)
IF (IF 1 THEN 2 ELSE 3) THEN 4

IF N(CODE)R = "2" THEN (N(QTY)*N(PRICE2)R)(MD2) ELSE IF N(CODE)R =
    "3" THEN (N(QTY)*N(PRICE3)R)(MD2) ELSE (N(QTY)*N(PRICE1)R)(MD2)
```

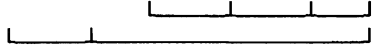
Normally, the code will comprise matching sets of IF/THEN/ELSE clauses, each ELSE paired with the most recent IF/THEN; however, the END keyword may be used for situations where an ELSE is not to be paired with the most recent IF/THEN:

```
IF 1=2 THEN IF 3=4 THEN 5 END ELSE 6
```



in which the ELSE 6 pairs with the first IF and the first THEN, not with the THEN 5. Without the END, the code would be interpreted as:

```
IF 1=2 THEN IF 3=4 THEN 5 ELSE 6
```



14.12 L : Length

The L code is used to test the **length** of the data field and take appropriate action. There are three forms:

- L0 will output the *length* of the data value. For example, a data value of SAND would output 4, and a data value of CEMENT would output 6.
- L4 will output the data only if the value is 4 characters in length, longer or shorter strings will output null. For example, a data value of SAND would output **SAND**, and a data value of CEMENT would output **null**.
- L6,10 will output the data if the value is 6, 7, 8, 9 or 10 characters in length. If the data is shorter than 6 characters or longer than 10 characters, then the code will output null. For example, a data value of SAND would output **null**, and a data value of CEMENT would output **CEMENT**.

You may find that this code is rarely used.

14.13 MC : Mask characters

The range of MC mask character codes perform a variety of character manipulation on the data. The following forms may be encountered. We illustrate the effect of some of these codes on either of the data values:

The card measures 4.5" by 6.25'

or:

the cat sat on the Mat. so did the dog.

MCA will return only alphabetic characters:

Thecardmeasuresby

MCAN

MCNA either of these will return only alphabetic and numeric characters:

The card measures 4.5 by 6.25

MC/AN

MC/NA either of these will remove alphabetic and numeric characters:

. " . '

MCDX

MCXD will convert decimal values to hexadecimal, and hexadecimal values to decimal.

MCL will convert all letters to lower-case:

the card measures 4.5" by 6.25'

MCN will return only numeric characters:

45625

MC/N will remove numeric characters:

The card measures ." by .'

MCP will convert all non-printable characters to full stops, and leaves printable characters as they are. The characters which are regarded as non-printable are those below ASCII 32 and those above ASCII 126.

MCS will capitalise the first word of each sentence in the data, leaving all other characters unchanged:

The cat sat on the Mat. So did the dog.

MCT will convert all words to capitals and lower-case letters:

The Card Measures 4.5" By 6.25'

MCU will convert all letters to upper-case:

THE CARD MEASURES 4.5" BY 6.25'

These codes are particularly useful for transforming string and character data which is being processed by a Basic program.

14.14 MD / ML / MR : Mask decimal values

The MD mask decimal code, and the associated MR and ML forms, are used for formatting and presenting numeric data such as numbers, prices

and costs. The code has the general form:

```
Mjn{.}{z}{x1}{c}
```

where:

j is one of D or L or R and indicates the manner in which the data is to be justified:

- D right-justified,
- L left-justified,
- R right-justified.

This justification is only of significance when the x_i element is also used.

n is a numeric digit specifying the number of decimal places which are to be printed.

. indicates that commas are to be inserted between thousands.

z indicates that leading zeroes are to be suppressed. A zero value will appear as null.

x.i indicates that the output is to be overlaid on a field of fill character x of length i . The fill character x will be one of:

- * to fill with asterisks,
- # to fill with spaces, or
- % to fill with zeros.

As Figure 14.3 shows, if the ML code is used, the field will be filled to the *right*, otherwise the field will be filled to the *left*.

c is a code specifying the way in which **negative** values are to be presented, and will be one of the following:

- c negative values are to be followed by CR.
- D positive values are to be followed by DB and the minus sign is to be dropped from negative values.
- E negative values are to be enclosed in angle brackets.
- M negative values are to be *followed by* the – sign.
- N the minus sign is to be suppressed on negative values.

The + sign is not shown on positive numbers.

The examples in Figure 14.3 illustrate the effect of some of these codes on the values 12345 and -12345 respectively.

Code	Value 12345	Value -12345
MD2	123.45	-123.45
MD2%10	0000123.45	000-123.45
MD%10	0000012345	0000-12345
MD,*10	****12,345	***-12,345
ML,*10	12,345****	-12,345***
MR,*10	****12,345	***-12,345
MD2C	123.45	123.45CR
MD2D	123.45DB	123.45
MD2E	123.45	<123.45>

Figure 14.3: MD / ML / MR codes

14.15 MT : Mask time values

Applications which need to record clock times usually hold these in the Advanced Pick **internal** format and represent the number of seconds since midnight, as illustrated in Figure 14.4.

Internal	External	Internal	External
0	00:00:00	9000	02:30:00
0	12:00AM	10000	02:46:40
15	00:00:15	10000	02:46:40
100	00:01:40	20000	05:33:20
1000	00:16:40	30000	08:20:00
2000	00:33:20	40000	11:06:40
3000	00:50:00	43200	12:00:00
3600	01:00:00	43200	12:00PM
4000	01:06:40	50000	13:53:20
5000	01:23:20	60000	16:40:00
6000	01:40:00	70000	19:26:40
7000	01:56:40	80000	22:13:20
8000	02:13:20	86399	23:59:59

Figure 14.4: Internal and external times

The MT code is used to convert such times between the internal form and any one of the following external formats:

- MT** will convert the internal time to a form such as **00:20** or **17:30** showing just the hours and the minutes.
- MTS** will convert the internal time to a form such as **00:20:34** or **17:30:59** showing the hours, minutes and seconds.
- MTH** will convert the internal time to a form such as **12:20AM** or **05:30PM** using the suffix AM or PM.

MTHS will convert the internal time to a form such as **12:20:34AM** or **05:30:59PM** showing the seconds and the suffix AM or PM.

14.16 P : Pattern matching

The **P** code is used to indicate that data values are only to be output if they match one (or more) of a specific set of **patterns**. The general form of the code is:

`P(pattern)`

OR:

`P(pattern){:(pattern)}`

with any number of *patterns* enclosed in parentheses and separated by semi-colons. The patterns consist of a sequence of elements such as:

7A which matches any string of 7 letters (upper- or lower-case),

3N which matches any string of 3 digits (0 to 9),

5X which matches any string of 5 characters (letters, digits, spaces or special characters).

0A which matches an empty (null) string or a string of any length made up of letters only,

0N which matches an empty (null) string or a string of any length made up of digits only,

Specific characters may be included in the pattern and are enclosed in apostrophes or quotation marks. These patterns are expressed as in other areas of Advanced Pick – such as Basic. Here are some examples of the **P** code:

P(3N) will output the data field if it consists of three digits. Values such as **000** or **111** or **987** will match; values such as **12** or **1234** or **1.1** or **111.1** or **1.111** or **ABC** will not match this pattern and will output a null value.

P(0A) will output the data field if it is null or consists only letters.

P(1NON) will output the data field if it consists of at least one digit and only numeric digits. Values such as **1** or **1029384756** will match; values such as **1A** or **1AB** will not match.

P(2A4N) will output the data field if it consists of two letters followed by four digits. Values such as **AA1111** or **ab1029** will match.

P(2A/'2N)

P(2A/2N)

These two are equivalent and will output the data field if it consists of two letters followed by a / and two digits. Values such as **AA/99** or **XZ/00** will match.

P(3N);(3A'-'4N)

will output the data field if it consists of *either* three digits (3N) or three letters followed by a hyphen and four digits. Values such as **333** or **AAA-9999** or **ABC-1029** will match.

If the data matches any of the specified patterns, then the data will be output, otherwise a *null* will be output.

14.17 R : Range checking

The R code is used to specify that a numeric data value is only to be output if it falls within one (or more) of a specified **range** of values.

R1,6 will test that the value of the data field is in the range 1 to 6 inclusive (= 1, 2, 3, 4, 5, 6).

The values 01, 001, 0001 will also be acceptable in this particular range.

R1,6;8,10 will test that the data field is *either* in the range 1 to 6 (= 1, 2, 3, 4, 5, 6) or in the range 8 to 10 (= 8, 9, 10).

If the data value is in any of the specified range(s), then the data will be output, otherwise a *null* will be output.

14.18 S : Substitute zero and null values

The S processing code is used to test the current data value and **substitute** another value according to whether or not the current data value is 0 (or null).

S;'NOT ZERO';'ZERO'

will test the current data and output NOT ZERO if the value is non-zero and non-null, otherwise it will output ZERO.

S;7;3 will test the current data and output attribute 7 if the value is non-zero and non-null, otherwise it will output attribute 3.

S;7;'VALID'

will test the current data and output attribute 7 if the value is non-zero and non-null, otherwise it will output VALID.

S;*;'ZERO'

will test the current data and output the value if it is non-zero and non-null, otherwise it will output ZERO.

These forms can be written more explicitly by means of the IF code.

14.19 T : Text substring extraction

The T code is used to pick out one or more characters from a data value. Let us imagine that the data value is:

FLOOR D, THE CHESTNUTS

then the codes:

T3,5 will return the 5 characters starting at the 3rd character of the data:
OOR D

T,5

T1,5 are equivalent and will return the first 5 characters of the data: FLOOR

T5 will return the **right-most** 5 characters (**TNUTS**) if the field is right-justified (that is, if there is R in attribute-type field of the attribute definition), otherwise ...

T5 will return the **left-most** 5 characters (**FLOOR**) of the field, if there is L or T or U in the attribute-type field of the attribute definition.

14.20 Tfile : File translation

The **Tfile** – translate file – processing code uses the current data value (let's call this the **source value**) as the key to another file and then reads (or **translates**) a specific attribute from that item on that file.

We can illustrate this by looking at items on the STOCK file which hold a **supplier code** (in attribute 6) and use this (the source value) to pick up data from the corresponding items on the SUPPLIER file. Two STOCK items are shown in Figure 14.5 (a), and these hold the supplier codes 2000 and 1000, respectively. In (b) we see the two items 1000 and 2000 on the SUPPLIER file. The definitions in (c) can then be used to output the original source value and the data translated from the SUPPLIER file, to produce the report shown in (d). The Tfile code:

TSUPPLIER;C;;1

in the attribute definition for SUPPLIER.NAME takes the source value (attribute 6) and uses this to read data from the SUPPLIER file and return attribute 1 of the corresponding SUPPLIER item.

1234		CHAIR, RED, OAK^7^DN/5/88^1000^30^2000^9557
9000		SIDEBORD, YELLOW, ASH^99^DN/14/70^13000^15^1000^9539

(a) The STOCK items

1000	ROGER PETERSEN PLC^081-999 4767
2000	A.B.ATKINS AND CO^061-999 2375

(b) The SUPPLIER items

000 SUPPLIER	SUPPLIER.NAME
001 A	A
002 6	6
003	
004	
005	
006	
007	TSUPPLIER;C;;1
008	
009 T	T
010 8	20

(c) Definitions on the DICT of the STOCK file

:SORT STOCK DESCRIPTION SUPPLIER SUPPLIER.NAME			
STOCK.... DESCRIPTION..... SUPPLIER SUPPLIER.NAME.....			
1234	CHAIR, RED, OAK	2000	A.B.ATKINS AND CO
9000	SIDEBOARD, YELLOW, ASH	1000	ROGER PETERSEN PLC

(d) The STOCK report

Figure 14.5: Tfile processing code

Here are some further examples of the code:

- TSUPPLIER;C;;2

will use the source value as the key to an item on the SUPPLIER file and then return attribute 2 of that item; using the data structure shown in Figure 14.5, this would be the telephone number. If there is no such item on the SUPPLIER file, or the item exists but attribute 1 is null, then the source value will be returned.
- TDICT PARMS;C;;7

will use the source value as the key to an item on the DICT section of the PARMS file and then return attribute 7 of that item. If there is no such item on the DICT section of the PARMS file, or the item exists but attribute 7 is null, then the source value will be returned.
- TSTOCK;X;;9

will use the source value as the key to an item on the STOCK file and then return attribute 9 of that item. If there is no such item on the STOCK file, then a null value will be returned.

TSTOCK;X2;;1

will use the source value as the key to an item on the STOCK file and then return the second multivalued attribute 1 of that item. If there is no such item on the STOCK file, then a null value will be returned.

TCLIENT;V;;3

will use the source value as the key to an item on the CLIENT file and then return attribute 3 of that item. If there is no such item on the CLIENT file or attribute 3 is null, then the original item will be omitted from the output.

The general form of the Tfile code is:

T{DICT }filename;subcode{x};;a

where *filename* identifies the file from which the data is to be read, *subcode* is one of the letters C or X or V, indicating the action to be taken if the required data *cannot be found* on the file; *x* is the number of the specific multivalued attribute which is to be returned or * if the multivalued attribute in the same position as the source multivalued attribute is to be returned; if *x* is omitted, then the entire data attribute will be returned (and any value-marks replaced by spaces); *a* is the number of the attribute which is to be read from the file. The full range of subcodes is:

- c** to translate and return the required attribute. If the item does not exist in the translate file, or the item exists but the specified attribute is null, then the original value is returned.
- x** to translate and return the required attribute, or to return a null value.
- v** to translate and return the required attribute, or to omit this item entirely from the report.

The full form of this most useful code is described in EPick and the reference literature.

14.21 Xc : Update Processor

Used in the correlative of a data-level identifier item, this form of the X code specifies an **update stamp** that is to be written to an item as it is updated. There are several forms of the code:

XAn

XAnV will store the user-id of the user performing the update in attribute *n* of the updated item.

XDn

XDnV will store the current date in attribute *n*.

XSn

XSnV will accumulate the number of seconds for which the item was handled by the Update Processor and store this in attribute *n*.

XTn

XTnV will store the current time in attribute *n*.

Combinations of two or more of these are acceptable. If the element *v* is included, then the stamp will be **appended** to attribute *n* as a new multivalued, otherwise the stamp will replace the previous contents of the attribute.

14.22 Other codes

There are a number of other, less important processing codes. We summarise these here:

MI used as an input-conversion, the **mandatory input** code specifies that there must be a data value in the field. The Update Processor will not allow you to file any item in which any such field is null.

MP converts **packed** decimal data to decimal, and vice versa.

MS specifies that data is to be sorted into other than the standard ASCII sequence. This will be used as a correlative in an attribute definition which is to be used with one of the BY / BY-EXP / BY-DSND / BY-EXP-DSND modifiers. The required sequence of characters must be specified as the single attribute of an item called SEQ which is set up on the MESSAGES file. Thus, there can only be one such sequence in effect at any one time.

MX converts ASCII data to hexadecimal format, and vice versa.

MY converts hexadecimal data to ASCII character format, and vice versa.

O used as an input-conversion, this code specifies that the multivalued of this field are to be sorted into left-justified ascending order when the item is filed.

Uxxxx passes control to a **user-exit** to process the data. A user-exit is a special Assembly language program supplied to perform some particular processing which is not generally available on Pick and Pick-like systems. Some examples are:

U1070 to return the check-sum for the current item.

U201E to return a count of the number of attributes in an item (as illustrated in Figure 12.6).

uc070;n to return the n th multivalue from an attribute.

v_n used as an input-conversion, this specifies that no more than n multivalues may be held in this field. The special form V0 prevents users from entering *any* data into the attribute. The Update Processor will beep if you attempt to enter more than the specified number of multivalues.

x used as an input-conversion, this code specifies that the field cannot be amended by means of the Update Processor.

xC{n} centres the output within a field of n spaces.

xI forces all further conversions to be handled as input-conversions.

xO forces all further conversions to be handled as output-conversions.

xR{n} produces a running total of this field on an Access report.

xT toggles all further conversions to be handled as input/output-conversions.

zA outputs the item in Assembly language listing format.

zC interprets the data value as a **zip code** (or **postcode**) and displays the corresponding city, state, zip code and country. This code uses the standard COUNTRY, STATE and ZCF files.

EPick gives full details of the form and action of these codes.

14.23 Summary

Processing codes are used as output-conversions within attribute definitions to control the appearance of the output data in Access reports and when displayed by the Update Processor, as correlatives within attribute definitions to specify the way in which the output value is to be derived for an Access report, as input-conversions with attribute definitions when used by the Update Processor to control the validation and verification of the data entered by the user, and as operands in ICONV and OCONV codes to convert the format of the data handled by a Basic program.

15

File indexing

In order to read an item from an Advanced Pick file, you must know the name of the file and the item-id of the specific item that you want. For example, the items on the STAFF file will be uniquely identified by the item-id, the employee number, those on the STOCK file might be identified by the part-number, and so on. To process a particular item on the STAFF file by means of the Update Processor, you might use a command such as:

```
UPDATE STAFF 1234
```

and to read the same item within a Basic program, you will use a statement of the form:

```
READ RECORD FROM STAFF, 1234
```

However, there may be an operational requirement to find an item on the STAFF file if you know only the employee's surname or the department in which he works or the post code of his home address. In such situations, this information – the surname, department, post code – is known as a **secondary key** to the file. Normally, such secondary keys will *not* be unique – there may be several employees with the same surname or in the same department or with the same post code. We have seen that you could issue an Access sentence such as:

```
LIST STAFF WITH NAME = "SMITH"
```

but this will have to read through *every* item on the file looking for the SMITHs; this would take a considerable time for a large file, and for a Basic program, the same requirement would be a long and complicated process. Advanced Pick allows you to create an **index** of secondary keys. This index is held in alphabetical order and your Basic programs can scan through the index, looking for a particular secondary key – SMITH – and then find the item-id – or item-ids – of the item(s) with that secondary key; having found the item-ids of all the SMITHs you can read these items to find the one you want. A further application is within the Update Processor where we can **cruise** through the index, looking at the items on the file using the specified field as a key into the index. We looked at this in Chapter 10.

15.1 Creating the index

Let's consider the index for the STAFF file. The main STAFF file is a regular Advanced Pick file, as shown in Figure 15.1 (a), and is handled just like any other file. The *index*, however, is held in frames of disk space and is constructed and maintained automatically by Advanced

Pick. A file may have any number of indexes, one for the surname, one for the surname and first name combined, one for the department, one for the post code, one for the telephone number, and so on. In fact, an index may be based upon any information derived from the item. Some analysts use the term **B-tree** for such an index.

In Figure 15.1 (a), we see a simple Access listing of the file; in (b), we create the index and then display the D-pointer to see the index details which have been added to attribute 8. In (d), we dump the frame occupied by the index to look at the physical form and see that the index contains the secondary key (the surname) and the relevant item-id(s), and the entries in the index are sorted into alphabetical order.

The *instructions* for deriving the index-key from the data in the item are specified as an A processing code in the CREATE-INDEX command:

```
CREATE-INDEX STAFF A1
```

and Advanced Pick stores these instructions together with the FID of the root of the index in attribute 8 the D-pointer for the data section of the file. In our example, these are:

```
I7870A1
```

showing that the root FID of the index which is based on the A processing code:

```
A1
```

starts at frame 7870. The index is held in frames of virtual memory, not on a Pick file.

The general form of the CREATE-INDEX command is :

```
CREATE-INDEX file.name code
```

where *file.name* is the name of the file for which the index is to be created, and *Acode* is the A processing code which is to be used. Here are some further examples:

```
CREATE-INDEX STAFF A1: '*':2(T1,1)
```

will create another index for the STAFF file with entries consisting of the surname (attribute 1), an asterisk and the initial of the given name (attribute 2).

```
CREATE-INDEX LIBRARY A0
```

will create an index to the LIBRARY file, the entries consisting of the item-id of the items on the file. This would be useful for cruising through the file in item-id sequence.

```
:SORT STAFF BY SURNAME SURNAME FIRST
```

```
STAFF..... SURNAME
```

```
FIRST.....
```

007	CLARK	PAUL
006	EDWARDS	JOHN
016	EDWARDS	PETER
014	JENKINS	DENNIS
015	KNIGHTON	ADAM
008	MURRAY	RHONDA
003	NAYLOR	ERNEST
001	PEARCE	PETER
020	QUIRK	HARRY
002	THOMAS	ANNE
009	THOMAS	ANDY

(a) The items on the STAFF file

```
:CREATE-INDEX STAFF A1
000011                               <= 11 items were added to the index

:LIST-ITEM DICT STAFF STAFF

      STAFF
001 D
002 7861
003 7
004
005
006
007
008 I7870A1                          <= Set up by the CREATE-INDEX process
009 L
010 10
```

(b) The index in the file D-pointer

```
:DUMP 7870

FID: 7870 : 0 0 7868 0 ( 1EBE : 0 0 1EBC 0 )

000 :.....CLARK^007__EDWARDS^006^016__JENKINS^01:
050 :4__KNIGHTON^015__MURRAY^008__NAYLOR^003__PE:
100 :ARCE^001__QUIRK^020__THOMAS^002^009^.....:
150 :.....:
```

(c) The raw index

Figure 15.1: B-tree index for the STAFF file

Having created the index, it is automatically maintained whenever items on the main data file are added, modified or deleted by Basic programs or any of the standard Advanced Pick tools, including:

- o The Update Processor,
- o The EDIT verb,
- o The COPY verb,

- The T-LOAD verb.

A number of Basic statements and TCL facilities are provided to support indexing: VERIFY-INDEX checks the integrity of an index; NFRAME-INDEX displays the number of frames occupied by an index; DELETE-INDEX deletes an index.

15.2 Using the index in Access

As Figure 15.1 illustrates, the entries in the SURNAME index are held in sorted sequence. If you issue a SORT or an SSELECT sentence and there is an index for the sort field which is used there, then the index will be used to speed up the action of the sort. Thus, if we issue an Access sentence such as:

```
SORT STAFF BY SURNAME
```

and the processing code in the attribute definition for the dataname SURNAME matches the A processing code(s) for the index(es) for the STAFF file, then this index will be used to retrieve the sorted data. There are other verbs which can exploit indexes: WLIST, WSORT, WSELECT and WSSELECT.

The DUPLICATE modifier may be used in conjunction with the WITH modifier in a context such as:

```
LIST STAFF WITH DUPLICATE SURNAME SURNAME FIRST
```

to list only those items which have duplicate index keys for the SURNAME field, that is, where two or more employees have the same SURNAME.

15.3 Using the index in Basic programs

The syntax of the Basic READ statement still requires a specific item-id:

```
READ RECORD FROM FILE, item.id ...
MATREAD ARRAY FROM FILE, item.id ...
READV VARIABLE FROM FILE, item.id ...
```

but the ROOT and KEY statements can be used to find an item-id for a given secondary (index) key and this item-id can then be used in a READ statement to get the item for processing. The sequence of events in using these statements is:

- 1) ROOT to find the root of the index to be used.
- 2) KEY to find the item-id (or nearest match) for a given secondary key.
- 3) READ to retrieve the true data item.

The Basic ROOT statement is first used to find the FID of the root of an index for a file and equip the program to access items by means of the index. This has a form such as:


```
ROOT 'STAFF', 'A1' TO ROOT.V ELSE STOP 'NO INDEX'
```

specifying the name of the file for which the index is to be used, and the A processing code on which the index is based. The variable – ROOT.V in this example – is the **index-variable** which is used to store the details of the root for use in subsequent KEY statements which will retrieve item-ids from the index.

The Basic KEY statement allows the program to access items by means of the index, by returning an item-id which matches the secondary key provided by the program. The form of the statement is:

```
KEY(CODE, ROOT.V, SURNAME, STAFF.ID) ELSE GO 100
```

where CODE holds any of:

- C to return the STAFF.ID which is the closest match to the SURNAME,
- N to return the next item-id to SURNAME,
- P to return the previous item-id,
- R to return the item-id of the exact match.

and ROOT.V is the **index-variable** set by means of the ROOT statement, SURNAME holds the secondary index key which is being sought, and STAFF.ID will return the item-id of the required item (as specified by CODE). The ELSE clause is executed if the required surname cannot be found.

The use of these statements is illustrated by the Basic program and the output shown in Figure 15.2.

15.4 The B-tree index

The technical reader will be interested in the physical organisation of a B-tree index. To illustrate this, let us consider a NAMES file for which we wish to create an index for attribute 1, the employee's name. When we issue the command:

```
CREATE-INDEX NAMES A1
```

the index will be constructed. If we inspect the data-level identifier for the file, we find that this looks like Figure 15.3 (a), indicating that the root of the index is at frame 24190. This root is essentially an index to the index. Dumping the root frame, we produce the display shown in Figure 15.3 (b).

```
* Demonstrate the ROOT / KEY statements
OPEN 'STAFF' TO STAFF.FV ELSE STOP
ROOT 'STAFF', 'A1' TO ROOT.V ELSE STOP
LOOP
  INPUT SURNAME
  UNTIL SURNAME='.' DO
    KEY('N', ROOT.V, SURNAME, ITEM.ID) THEN
      CRT ITEM.ID
```

```

        READ REC FROM STAFF.FV, ITEM.ID THEN
            CRT REC
        END ELSE
            CRT 'Not found'
        END
    END ELSE
        CRT 'No ':SURNAME:' on STAFF file'
    END
REPEAT
```

```

Surname: EDWARDS
006
EDWARDS`JOHN
Surname: EDWARDS
016
EDWARDS`PETER
Surname: J
014
JENKINS`DENNIS
```

Figure 15.2: Using an index in a Basic program

```

NAMES
01 D
02 26354
03 17
04
05
06
07
08 I24190A1
09 L
10 10
```

<=== Root FID: 24190

(a)

```

FID: 24190 : 0 24197 26371 0 ( 5E7E : 0 5E85 6703 0 )

0000 01000000 0000855E 080046FE 0000885E 000 :.....^..F^...^:
0010 10004E45 56498C4C 45204CFE 00008E5E 016 :..NEVILLE L^...^:
0020 10005341 59455220 4EFE0000 0000875E 032 :..SAYER N^.....^:
```

(b)

Figure 15.3: The root frame

If we look at the fields which I've underlined, we find that:

- Names before F are indexed in frame .5E85 (that is 24197 in decimal). Note that 5E85 is held as 855E
- Names F and above but before NEVILLE L in frame .5E88
- Names NEVILLE L and above but before SAYER N in frame .5E86
- Names SAYER N and above in frame .5ECF.

This is confirmed if we dump the appropriate frames, as illustrated in

Figure 15.4.

```
FID: 24197 : 0 24200 24190 0 ( 5E85 : 0 5E88 5E7E 0 )
000 : .....ABRAHAMS P^126^151 ..BAILEY E^150 ..BAILEY:
050 : I^122 ..BAILEY S^123 ..BARRETT B^087 ..BARRETT:
100 : F^093 ..BARRETT H^035 ..BARRETT K^020^095 ..BROWN :

500 :017 ..EDWARDS K^025 ..EDWARDS L^049 ..EDWARDS R^14:
550 :6 ..EISENBACH F^074 ..EISENBACH O^065 ..EISENBACH :
600 :W^071 ..ELLISON C^117 ..ELLISON F^114 .......:

FID: 24200 : 0 24198 24197 0 ( 5E88 : 0 5E86 5E85 0 )
000 : .....FARMER F^135 ..FEATHERSTONE F^111 ..FEATHE:
050 : RSTONE S^033 ..GARDINER J^083 ..GAUNT G^070 ..GR:
100 : EENFIELD J^054 ..HACKMAN D^060 ..HALL R^097 ..HA:

550 :LLORY A^056 ..MANNERS J^077 ..MANNERS T^107 ..NAYL:
600 :OR N^009 ..NEVILLE D^045 ..NEVILLE G^040 ..NEVILL:
650 :E K^086 .......:

FID: 24198 : 0 24271 24200 0 ( 5E86 : 0 5ECF 5E88 0 )
000 : .....NEVILLE L^118 ..NEVILLE P^133 ..OLDFIELD A^1:
050 :06 ..OLDFIELD F^022 ..PARKER D^075 ..PARKER J^1:
100 :36 ..PARKER T^108 ..PRITCHARD A^104 ..PRITCHARD :

400 :J^002 ..SALE F^015^044^124 ..SALE V^142 ..SALTER:
450 :A^041 ..SALTER W^066 ..SAYER G^149 ..SAYER K^11:
500 :9 .......:

FID: 24271 : 0 0 24198 0 ( 5ECF : 0 0 5E86 0 )
000 : .....SAYER N^001 ..SAYER W^084 ..SCOTT A^115 ..SC:
050 :OTT J^007^138 ..SCOTT K^125 ..SCOTT L^064 ..SCOTT :
100 :S^031 ..SCULLY A^027^141 ..SCULLY D^112 ..SCULLY:

400 :^131 ..VALDEZ F^101 ..VALDEZ M^089 ..WALCOT F^0:
450 :37 ..WALCOT J^043 ..WALCOT S^016 ..WARDLE A^006:
500 : ..WARDLE G^052 ..WRIGHT L^109 .......:
```

Figure 15.4: The leaf frames of the index

We can visualise the index as shown in Figure 15.5.

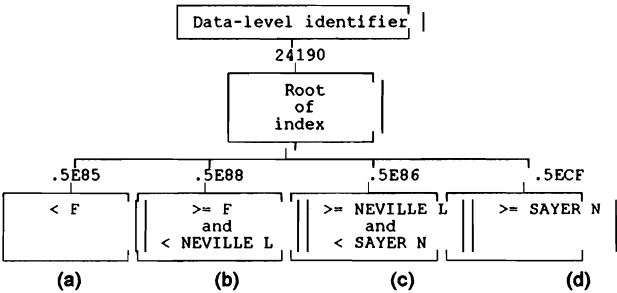


Figure 15.5: The organisation of the B-tree – 1

Such a data structure is known as a **B-tree** – a **balanced tree** – with a **root frame** from which are suspended one or more **leaves** or **leaf frames**. The routines which maintain the index attempt to distribute the names through the leaf frames in a **balanced** manner, with each individual leaf occupying no more than one frame.

To illustrate what happens as the index expands, let us consider that we have added a number of new employees called **NOLAN** and others called **NORMAN**. These will be added – one by one – to leaf (c), and as

the frame is filled it will be split into two. The other leaves are unchanged. The root frame now looks like that in Figure 15.6, giving us a diagram like that in Figure 15.7.

```
FID: 24190 : 0 24197 26371 0 ( 5E7E : 0 5E85 6703 0 )

0000 01000000 0000855E 080046FE 0000885E 000 : .....^..F^...^:
0010 10004E45 56497C4C 45204CFE 0000865E 016 : ..NEVILLE L^...^:
0020 10004E4F 524D414E 2056FE00 00002757 032 : ..NORMAN V^...^g:
0030 10005341 59455220 4EFE0000 0000075E 048 : ...SAYER N^...^:
```

Figure 15.6: The root frame – 2

The first two leaves and the last are unchanged, but the entries in .5E86 have been split into two, as illustrated in Figure 15.8.

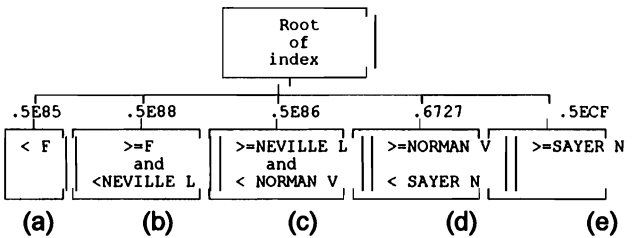


Figure 15.7: The organisation of the B-tree – 2

There are several noteworthy points, some of which are verified by the above illustrations:

- The root frame is a single frame in this example. If the number of entries in the root frame increases, the root will be split into two frames with the new root showing the distribution of the two frames.
- The entries in the root frame show the first key in each frame. It is not always necessary to show the full key; in fact, Advanced Pick uses the shorted substring which distinguishes the first key in one frame from the final key in the previous frame.
- The entries in the index comprise the index key (as specified by the A code when the index is created) followed by an attribute-mark followed by the item-id (or item-ids) of the item(s) for which that is the index key.
- A control field showing the length of the index entry precedes each index entry.
- Each entry is followed by one or two segment marks, so that the entry is an even number of bytes in length.
- There is only one level of leaf frames.
- Each leaf occupies a single frame.

- As a leaf grows, it is split into two leaves, each occupying no more than one frame and each holding approximately the same number of entries.
- The entire B-tree is constructed and linked together as a group of frames, starting at the frame which holds the data-level D-pointer for the file, then the root frame, then the leaves in order.

```

FID: 24198 : 0 26407 24200 0 ( 5E86 : 0 6727 5E88 0 )

000 :.....NEVILLE L^118 ..NEVILLE P^133 ..NOLAN A^178 :
050 :..NOLAN B^189 ..NOLAN C^187 ..NOLAN D^180 ..NOLAN :
100 :E^173 ..NOLAN F^181 ..NOLAN G^182 ..NOLAN H^183 ...:

500 : L^166 ..NORMAN M^171 ..NORMAN N^170 ..NORMAN P:
550 :^157 ..NORMAN R^154 ..NORMAN S^159 ..NORMAN T^1:
600 :55 .....:

FID: 26407 : 0 24271 24198 0 ( 6727 : 0 5ECF 5E86 0 )

000 :.....NORMAN V^168 ..NORMAN W^152 ..OLDFIELD A^1:
050 :06 ..OLDFIELD F^022 ..PARKER D^075 ..PARKER J^1:
100 :36 ..PARKER T^108 ..PRITCHARD A^104 ..PRITCHARD :

400 :J^002 ..SALE F^015^044^124 ..SALE V^142 ..SALTER:
450 : A^04I ..SALTER W^066 ..SAYER G^149 ..SAYER K^11:
500 :9 .....:

```

Figure 15.8: A frame is divided

15.5 Summary

Any file may have one or more B-tree indexes to facilitate the retrieval of items by way of secondary keys. A number of TCL facilities and Basic statements are provided to use such indexes. Once they have been created, indexes are automatically maintained whenever items are added to, changed, or deleted from the file.

16

The Basic language

At most Advanced Pick installations, the data processing is performed by purpose-built programs written and supported by a team of programmers. At other sites, the applications are constructed with the aid of system generation tools, such as *ALL*, *Creator*, *Libra*, *SB+*, and *System Builder*, supplemented by Basic programs and subroutines. In all cases, the **Basic language** is used as the programming medium.

The dialect of Basic which is offered on the Pick system is a very powerful version of the original **Dartmouth Basic** which has been extended to allow programs to handle Pick data files and other aspects of the system. **Pick/Basic** has further features, including:

- **Optional statement labels** and **alphanumeric statement labels**.
- Several statements may be written on one line.
- **Meaningful variable-names** may be used, and variable-names may be of any length.
- Any variable may hold either numeric data or an alphanumeric string.
- Output may be sent to the screen or to the **spooler**, that is the printer.
- Reports may be produced on several different spooler queues at the same time.
- **Structured programming facilities**: FOR ... NEXT statement; LOOP statement; CASE statement; IF ... THEN ... ELSE statement; Computed GOTO and computed GOSUB statements.
- **Multiline structures** for IF statements and certain other statements.
- **Modular programming facilities**: internal subroutines, using GOSUB and RETURN statements; external subroutines using CALL and SUBROUTINE statements.
- **External subroutines** can be written as independent processing modules used by one or more programs.
- Use of a **COMMON** data block to pass **global information** between modules.
- **File handling facilities**: OPEN, READ, WRITE and DELETE statements.
- **Direct access** to file items by their item-id, or indirectly by a secondary key maintained by the Advanced Pick indexing features.
- **File-locking** to prevent several users updating the same record at the same time.
- Handling **backing storage devices**: magnetic tape, floppy diskette.

- Handling the data structure of Advanced Pick items: attributes, values and subvalues.
- **Dynamic arrays:** this is a string of data which is viewed as a sequence of elements. There may be any number of elements, and special functions support the deletion and insertion of elements, searching for a specific element, and sorting into sequence as elements are added to an array.
- **Debugging software** to simplify program development. We discuss this in Chapter 18.
- A wide range of **intrinsic functions** for general data processing and for obtaining information about the system and the working environment.
- Ability to invoke any TCL commands or Access sentences from within a program, and to pass the output back to the program for further processing.

So let's take a look at Advanced Pick Basic. I'll use the term **Basic** here, or, if I need to distinguish this from other dialects of the language, I'll use the term **Pick/Basic**. The **Proc** language, the other programming medium for Advanced Pick systems, is discussed in Chapter 17. The **Advanced Pick Assembly language** is a specialised topic and is sold separately from the standard system; we shall not discuss the Assembly language in this book.

So let's take a look at Advanced Pick Basic. I'll use the term *Basic* here, or, if I want to distinguish this from other dialects of the language, I'll use the term *Pick/Basic*. The *Proc language*, the other programming medium for Advanced Pick systems, is discussed in Chapter 17. The *Advanced Pick Assembly language* is a specialised topic and is sold separately from the standard system; we shall not discuss the Assembly language in this book.

If you have used any other programming language, such as C, Fortran, Pascal, or even Cobol, you will find that Basic has few shocks and a few surprises. If you have used Basic on any other system, you will be delighted to see how wonderfully Pick/Basic has been adapted to meet the needs of the commercial programmer. If you have used Pick/Basic in any other implementation, you may find it sufficient to read the notes on the differences which I presented in Chapter 2.

16.1 Basic programs

Let's start by looking at the Basic program shown in Figure 16.1.

```
01 * Program to construct simple Access sentences
02
03     OPEN 'DICT STOCK' TO STOCK.DICT ELSE STOP
04
```

```
05  PROMPT ''
06
07  EQUATE DN TO CHAR(10)
08  MSG1 = DN : 'Enter dataname #'
09  MSG2 = DN : 'Cannot find dataname '
10  MSG3 = 'Enter a STOCK dataname or . if no more'
11  COUNT = 1
12  SENTENCE = 'SORT STOCK'
13
14  CRT @(-1) : @(25,0) : '** Access sentence builder **'
15
16  LOOP
17      GOSUB Get.dataname
18  UNTIL DATANAME = '.' DO
19      GOSUB Vet.dataname
20  REPEAT
21      GOSUB Execute.sentence
22  STOP
23
24 * Subroutines *****
25
26 Get.dataname:
27
28  CRT MSG1 : COUNT : ' ':
29  INPUT DATANAME
30  DATANAME = TRIM(DATANAME)
31  RETURN
32
33 Vet.dataname:
34
35  READ DEFN FROM STOCK.DICT,DATANAME ELSE GO MISSING
36  IF DEFN<1> NE 'A' AND DEFN<1> NE 'S' THEN GO MISSING
37 Add.dataname:
38  SENTENCE = SENTENCE : ' ' : DATANAME
39  COUNT = COUNT + 1
40  RETURN
41 MISSING:
42  CRT MSG2 : DATANAME
43  CRT MSG3
```

```

44      RETURN
45
46 Execute.sentence: *****
47
48      CRT DN : 'Executing ' : SENTENCE
49      CRT 'Proceed? Y/N? ' :
50      INPUT RESP,1
51      IF RESP='Y' THEN EXECUTE SENTENCE
52      RETURN

```

Figure 16.1: A Basic program

Each Basic program is created and held as an item on a file; the individual **statement lines** of the **source program** are the **attributes** of the **program item**. You can hold your programs on any file – provided that it has a DICT section and a data section – and it is the usual practice to keep all the programs for a particular application together on one file.

Let's imagine that I've created the program shown in Figure 16.1 as an item on the file MBPROGS and saved it with the item-id SENTENCE01. The program would be created and changed by means of either the Update Processor or the Pick Editor:

```

UPDATE MBPROGS SENTENCE01
OR:
EDIT MBPROGS SENTENCE01

```

Before it can be executed, the Basic program must be **compiled** by means of one of the commands:

```

BASIC MBPROGS SENTENCE01
OR:
COMPILE MBPROGS SENTENCE01

```

In this particular case, I could use either of these commands to compile the program. If I had typed the keywords in *lower-case*:

```

open 'dict stock' to stock.dict else stop
loop if resp='y' then execute sentence

```

and so on, then I must use the COMPILE verb, which will accept the keywords in lower-case. The compiler translates the Basic statements of the source program into an equivalent sequence of statements in the Pick Assembly language – the **object program** and saves this as the item SENTENCE01 on the DICT section of the MBPROGS file. Strictly speaking, this is not true **object code** since it is not the **machine code** for the computer which I'm using and it cannot be executed directly by the computer. Instead, this **pseudo-object code** is interpreted into true machine code, statement by statement, when I execute the program by

means of the command:

```
RUN MBPROGS SENTENCE01
```

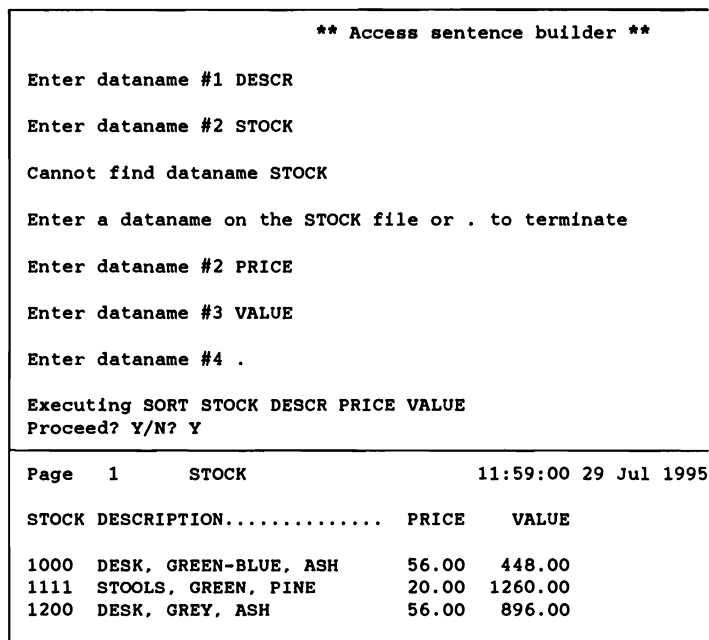


Figure 16.2: Program output

The action of the program shown here is to ask the user for a series of datanames for the STOCK file, check that there is an attribute definition for each dataname, and then use these to construct an Access sentence; finally, the sentence is executed. Figure 16.2 shows a typical screen display which might be produced when this program is executed. I've only shown the first part of the Access report which is produced.

Even if you haven't done any programming before, you can probably understand some of the Basic program, and with the help of Figure 16.2 you may be able to figure out what the program is doing. For those new to Basic and programming, let us look at some of the statements which I have used in this program, noting any interesting points:

- The numbers to the left of the program statements in Figure 16.1 (01 to 52) are the **line-numbers** of the individual statement lines in the program; these are the 52 attributes of the program item. The line-numbers only appear when we display the source program and are not a part of the text of the program itself. This is where Pick/Basic is quite different from those dialects of Basic which need line-numbers to organise the statements into the correct order.
- The lines which begin with an asterisk are **comments** and I include these to explain what the program is doing. The computer ignores

these comments when it is executing the program.

- o **Blank lines** may be included within the program to make it more readable.
- o Some lines begin with a **statement label** such as:

```
Get.dataname
MISSING
```

These serve as *destinations* when I want to jump about the program with GOSUB, GOTO and other statements. You can have numeric statement labels, or alphanumeric label, as here. Alphanumeric labels must be followed by a colon.

- o The CRT statement displays information on the screen. I could have used the PRINT statement instead of CRT. The statement:

```
CRT 'Proceed? Y/N? ' :
```

displays the message:

```
Proceed? Y/N?
```

on the screen. The final *colon* holds the cursor in position immediately after the text.

- o The statement:

```
CRT @(-1) : @(25,0) : '** Access ...
```

uses the special codes: @(-1) clears the screen, and @(25,0) jumps to column 25 on line 0 (the opt line of the screen), and then display the text ** Access ...

- o The statements:

```
GOSUB Get.dataname
GOSUB Vet.dataname
GOSUB Execute.sentence
```

jump to the specified label and continue from there. When a RETURN statement is encountered, the processing *jumps back* to (the statement immediately after) the GOSUB statement. This arrangement of coding is known as a **subroutine**.

- o The statement:

```
INPUT DATANAME
```

waits for the user to type in some information, and when the user hits the <Return> key, that information will be placed in a **variable** identified by the name DATANAME.

- o The statement:

```
INPUT RESP,1
```

waits for the user to type in one character of information, and places this in a variable called RESP.

- The statement:

```
IF RESP='Y' THEN EXECUTE SENTENCE
```

tests whether the contents of the variable RESP (which has just been typed in by the user) is the letter Y, and, if so, the sentence which has been built up in the variable SENTENCE will be executed. The program has constructed an Access sentence in the variable SENTENCE, adding each dataname required by the user to the end of the sentence. If RESP does not hold the letter Y, then the processing will pass on to the statement on the following line, line 52.

Other interesting statements and structures are:

```
OPEN 'DICT STOCK' TO STOCK.DICT ELSE STOP
```

open the DICT section of the STOCK file and store the location and the modulo of the file in the variable STOCK.DICT.

```
LOOP ... UNTIL DATANAME='.' DO ... REPEAT
```

define a sequence of statements which is to be repeated until the user enters the character . as a dataname.

STOP terminate the execution of the program and, in this case, prevent the processing crashing into the subroutines at the end of the program.

```
DATANAME = TRIM(DATANAME)
```

take the dataname typed in by the user and remove any superfluous spaces.

```
READ DEFN FROM STOCK.DICT,DATANAME ELSE GO MISSING
```

use the contents of DATANAME as an item-id to read an item from the file which has been opened as STOCK.DICT. If there is no such item, then the processing jumps to the statement label MISSING; if there is an item, then the contents of that item are placed in the variable DEFN.

```
IF DEFN<1> NE 'A' AND DEFN<1> NE 'S' THEN GO MISSING
```

check attribute 1 of the data which has been read into DEFN to see whether this is an A or an S: if not, then the processing jumps to the label MISSING; if the first attribute of DEFN is A or S, then the processing continues to the next statement.

```
SENTENCE = SENTENCE : ' ' : DATANAME
```

append a space and the new dataname to the end of the sentence which is being built up in SENTENCE.

```
COUNT = COUNT + 1
```

add 1 to the contents of variable COUNT, keeping a count of how many datanames the user has typed in.

Some points which might interest experienced programmers are:

- The indentation of the program statements is entirely optional. I have used a scheme which attempts to show the structure and the dependence of statements such as the LOOP / REPEAT structure.

- Pick/Basic variables such as STOCK.DICT, COUNT, DEFN and so on, are not declared before they are used.
- The variable names and the statement labels can be as long as required. This allows you to produce readable coding and reduces the need for explicit comments.
- Pick/Basic makes no distinction between the names of variables which hold **numeric** data (such as COUNT) and those which hold **alphanumeric** data (such as SENTENCE).
- Since Advanced Pick is normally **case insensitive**, I need only check for a response of Y and this will handle the case where the user has entered y.
- Pick/Basic does not demand an END statement at the physical end of the program. Indeed, it is wise *not* to include a final END statement as this may hide missing END statements within the program.

Now let us look more closely at some of the features of the Basic language and Basic programming on Advanced Pick.

16.2 Variables, literals, numbers and strings

Two types of data are handled by Basic programs: **numbers** and **strings**. We can see these in action in this sequence:

```
PRODUCT = 'CHAIR'
DISCOUNT = 25.0
INPUT PRICE
COST = PRICE * (100-DISCOUNT) / 100
PRINT PRODUCT,COST
COUNT = COUNT + 1
```

Here, **numbers** (such as 25.0 and 100 and 1) and **strings** or **alphanumeric strings** (such as CHAIR) are encountered in two contexts: firstly, as explicit values – known as **literals** – which are coded into the program:

```
'CHAIR'
25.0
100
```

and secondly as values held in the variables PRODUCT, DISCOUNT, PRICE and COST. These are names which the programmer has used to identify a sequence of storage locations – the **variables** – which are available to a program as it is executing. The actual data which is held any particular variable will change as the program executes. The contents of a variable may be changed by an assignment statement such as:

```
PRODUCT = 'CHAIR'
PRODUCT = 'TABLE'
DISCOUNT = 25.0
DISCOUNT = 0
```

by a calculation:

```
COUNT = COUNT +1
COST = (PRICE * QUANTITY) * (100 + VAT.RATE) / 100
```

or by asking the user to type a new value into the variable:

```
INPUT PRICE
```

As a new value is assigned to a variable, any value already held there will be lost. Any variable may hold just one value and this may be a number or a string.

A **number** is a sequence of numeric digits with an optional arithmetic sign (– or +) and with an optional decimal point. Some valid literals and invalid literals are shown in Figure 16.3 (a). We discuss the available range of numbers later.

A **string** is a sequence of alphanumeric characters, and when used in a Basic statement, a string will be enclosed in " or ' or \ characters. Some literals are shown in Figure 16.3 (a). A string may be any length from 0 (for a null string) up to 2 gigabytes, that is, 2,147,483,648 characters in length.

Valid	Invalid	Valid	Invalid
25.0	+	'abcd'	"abcd'
+25.0	--123	"abcd"	'abcde'fghij'
-1000.3746	123..	\abcd\	\yes
100	123.0.	"PLEASE WAIT"	
0	123.456-	"1234"	
234.567	123.50CR	" "	
1234.	1,234	"12.34"	
0.001	£1000	\A\	
0.0		' '	
-00		' ----- '	
+0.0		"Can't find it"	
0000		"The name is:"	

(a) Numeric literals

(b) String literals

Figure 16.3: Literals

Unlike most other programming languages, Advanced Pick Basic will let you use numeric strings in arithmetic operations. Thus, if we consider the two statements:

```
COST = PRICE * (100-DISCOUNT) / 100
COST = PRICE * ("100"-DISCOUNT) / "100"
```

The first uses a **numeric literal** 100 and the second uses a **string literal** "100", but both are acceptable and both will give the same result. The Basic run-time processor recognises that we are doing arithmetic (because we use the arithmetic signs * for multiplication, – for subtraction and / for division), and will convert all the data to numbers. Because of

the need to convert the data, the second statement would be the slower. Obviously, it would be an error if we had written:

```
COST = PRICE * ("YES"-DISCOUNT) / "100"
```

but this non-fatal error would only be detected when the statement was executed and the run-time processor attempted to convert "YES" to a number, and it would, in fact, then use a value of 0. Similarly, if we were to use numeric literals in a string handling operation such as:

```
ADDRESS = 34 : NAME : ' STREET'
PRINT ADDRESS
```

the Basic run-time processor would convert the numeric literal 34 to a string because we are performing the string operation of concatenation (implied by the : sign), and then output a string such as:

```
34 HIGH STREET
```

16.2.1 Variables

Your program may use up to 214,748,362 variables at any one time (this includes individual named variables and all elements of any dimensioned arrays in the program), and each variable and array is uniquely identified by name. The name may be up to 48 characters in length, and must start with a letter followed by any combination of letters, digits, full stops, dollar signs and (on later releases) the underscore character.

Some acceptable and some unacceptable variable names are shown in Figure 16.4. The compiler will reject any attempts to use certain *reserved words*, or the names of the intrinsic Basic functions (a list of these is shown in the Appendix) for your arrays, or those keywords shown in Figure 16.4 (c) for your variables.

Data.X	NextR	STORE.32.A	TAX
DATAx	Next1	Subroutine9	TOTAL.TO.DATE
End1	RECORD	TABLE.B2	VALUES\$
Income.tax	SIZE.\$\$	Tax	VALUE1

(a) Acceptable names

\$PRICE	OQUANTITY	78.WIDTH	LAST NAME

(b) Some unacceptable names

AND	END	LOCKED	RETURNING
BY	FOR	NEXT	SETTING
CAPTURING	GO	ON	THEN
CASE	GOSUB	OR	TO
ELSE	GOTO	REPEAT	WITH

(c) Reserved words

Figure 16.4: Variable names


```
OAP=AGE>65
OAP=AGE>=65 AND SEX='M'
OAP=(AGE>=65 AND SEX='M') OR (AGE>=60 AND SEX='F')
```

```
IF OAP THEN CRT 'This employee is a pensioner'
IF NOT(OAP) THEN CRT 'Not eligible for pension'
IF OAP THEN CALL PENSION.ROUTINE
```

16.2.2 Precision

When a program execution begins, each variable is allocated a storage space of six bytes. When this is used to store numeric data, the six bytes (48 bits) are used to hold the number as a binary value. Allowing one bit for the arithmetic sign, this means that we can hold values in the (decimal) range $\pm 2^{47}-1$ that is ± 140737488355327

```
A=3540705968149597751242378595390670323015412790761
B=354070596814870000000000000000000000000000000000000
C=4371241896208856100100048221092623586370756606568
81926429
n=0
```

Figure 16.5: Large numbers

In some circumstances, there may be overflow or loss of precision if a value goes outside this range. However, if a calculated value exceeds this range, Advanced Pick holds the number as a **numeric string**. Numeric strings are unlimited in length and may have up to 18 places of decimals. Figure 16.5 illustrates these points; here we see how a calculated value is virtually unlimited in length and precision, although the exponentiation function 123456789^6 loses precision, and 123456789^7 returns zero.

Numeric values are normally held with *4 places of decimals*, so that a statement such as:

```
PRINT 2/3
```

will display the value:

```
0.6666
```

The PRECISION statement can be used to change this numerical accuracy from this default precision of 4 to any value in the range:

```
PRECISION 0
to:
PRECISION 9
```

There can only be one PRECISION statement in any program or subroutine, and this precision applies to all numbers throughout the program. A main program and its external subroutines must all be compiled with the same precision, although the standard functions are not affected by any precision imposed on a program or subroutine which uses them.

16.3 Dimensioned arrays

An **array** – sometimes called a **dimensioned array** – is a sequence of variables, each of which is identified by the name of the array and its position within the sequence. Basic will let you handle *one-dimensional arrays*, like INCOME in Figure 16.6 (a), and *two-dimensional arrays*, like SCORE in Figure 16.6 (b). Before you use such arrays, they must be **declared** explicitly in order to indicate how big they are and, consequently, how much storage the compiler is to allow. For example, a statement such as:

```
DIMENSION INCOME(12)
```

or the abbreviated form:

```
DIM INCOME(12)
```

would set aside 12 variables for an array called INCOME, and either of the statements:

```
DIMENSION SCORE(4,5)
```

```
DIM SCORE(4,5)
```

would set aside a table or a matrix of 20 variables for an array called SCORE. You might visualise these as in Figure 16.6.

Within the program, the individual elements of a one-dimensional array are identified by references such as:

- INCOME(1) for the first element in INCOME, which contains the number 33,
- INCOME(2) for the second element, which contains the string QQ, and so on up to ...
- INCOME(12) for the final element in INCOME, which contains the number 31.

and, for a two-dimensional array, by references:

- SCORE(1,1) for the top-left element of SCORE, which contains the number 77,
- SCORE(3,1) for the element which contains the letter X, and so on up to ...
- SCORE(4,5) for the bottom-right element of SCORE, which contains the number 55.

INCOME

33	QQ	AA	xx	yy	zz	19	51	66	69	45	31
1	2	3	4	5	6	7	8	9	10	11	12

(a) A one-dimensional array : a *vector*

SCORE

1	77	A	B	C	D
2	J	K	L	M	N
3	X	Y	Z	A	B
4	32	W	a	DR	55
	1	2	3	4	5

(b) A two-dimensional array : a *matrix*

SCORE

77	A	B	C	D	J	K	B	32	W	a	DR	55
1,1	1,2	1,3	1,4	1,5	2,1	2,2	3,5	4,1	4,2	4,3	4,4	4,5

(c) A two-dimensional array – physical organisation

Figure 16.6: Dimensioned arrays

The number(s) in parentheses which indicate the position of the element is called the **subscript**. These subscripted references are used in exactly the same context as are simple variables:

```
INCOME(1) = 0
INCOME(3) = INCOME(1) + INCOME(7)
INPUT INCOME(5)
PRINT INCOME(6)
INCOME(9) = 2 * INCOME(9)
PRINT SCORE(1,1)
SCORE(2,3) = SCORE(4,5)
INCOME(1) = SCORE(4,5) * 100
```

As the examples illustrate, the various elements of an array may contain different types of data; any element may contain a number and any may contain a string. Each element is handled just as if it were an independent variable. Figure 16.6 (c) shows the actual physical organisation of a two-dimensional array.

```
* Program to accept a set of numbers and calculate
* the mean and the standard deviation
```

```
PRECISION 6
EQUATE MASK TO '6'
CLEAR
DIMENSION NUMBER(100)
FOR X=1 TO 100
    CRT 'Enter value #':X:': ':
    INPUT NUMBER(X)
    IF NUMBER(X)='.' THEN
        X=X+1
        EXIT
    END
    COUNT=COUNT+1
    TOTAL=TOTAL+NUMBER(X)
NEXT X
CRT '          Values: ':COUNT
CRT '          Total: ':TOTAL MASK
MEAN=TOTAL/COUNT
CRT '          Mean: ':MEAN MASK
FOR C=1 TO COUNT
```

```
        DIFF=MEAN-NUMBER(C)

        DIFF.SQ=DIFF*DIFF

        SUM.SQ=SUM.SQ+DIFF.SQ

    NEXT C

    VARIANCE=SUM.SQ/COUNT

    CRT '          Variance: ':VARIANCE MASK

    STD.DEV=SQRT(VARIANCE)

    CRT 'Standard deviation: ':STD.DEV MASK
```

```
-----

Enter value #1: 20.4
Enter value #2: 20.9
Enter value #3: 20.2
Enter value #4: 21.8
Enter value #5: 21.8
Enter value #6: 20.8
Enter value #7: 21.8
Enter value #8: 20.9
Enter value #9: 20.2
Enter value #10: 20.1
Enter value #11: .

        Values: 10

        Total: 208.900000

        Mean: 20.890000

        Variance: 0.430900

Standard deviation: 0.656429
```

Figure 16.7: Using an array

The great power of an array comes from the ability to use a *variable* or an *arithmetic expression* as the subscript. For example, we might use a statement such as:

```
PRINT INCOME(X)
```

which would take the numeric value held in the variable X (let's imagine this contains the number 3), and use this subscript to print the contents of INCOME(3) the third element in the array. Figure 16.7 shows how we might use an array to store a series of numbers and then manipulate the

array to calculate the mean of the values and the statistical *standard deviation*. Some other examples of subscripts:

```
TOTAL = INCOME(VALUE)
ANSWER = SCORE(ROW,3)
INCOME(VALUE/2) = NEW1
SCORE(ROW,COLUMN) = INCOME(ROW*COL/2)
PRINT SCORE(INCOME(1),INCOME(C))
```

Array names conform to the same rules as Basic variable names, they must be unique within the program, and they must not conflict with the names of variables or any of the standard Basic functions. There may be any number of DIMENSION statements in a program, and several arrays may be declared on a single DIMENSION statement:

```
DIMENSION INCOME(12), SCORE(4,5), STOCK(10)
OR:
DIM INCOME(12), SCORE(4,5), STOCK(10)
```

An array may be dimensioned at run-time, using sequences like this:

```
SIZE=10
DIMENSION DIMARR(SIZE)
OR:
INPUT PEOPLE
DIMENSION NAMES(PEOPLE)
INPUT TERMS
DIMENSION SCORE(PEOPLE,TERMS)
```

If an array has been declared as, say:

```
DIMENSION INCOME(12)
```

then the first element of the array is INCOME(1) and the final element is INCOME(12); any subscripts outside this range, such as:

```
INCOME(0)
INCOME(100)
INCOME(-5)
```

are invalid and will cause a fatal compilation error (if a literal subscript is used) or a fatal execution error (if the subscript is calculated by the program). Non-integer subscripts are truncated, so that a reference to INCOME(1.5) would be interpreted as INCOME(1).

The MAT statement will allow you to assign values to all the elements of an array. For example:

```
MAT INCOME=1
    assign the value 1 to every element in the array INCOME.
```

```
MAT ISTORE=MAT INCOME
    copy the contents of array ISTORE into the array W, element by
    element. Clearly, care must be taken with the shape and size of the
    two arrays.
```

Beyond these forms, the MATPARSE and MATBUILD statements (described later), and the file handling statements:

```
MATWRITE ISTORE ON STATS.FV, TABLE.NAME
MATREAD INCOME FROM STATS.FV, TABLE.NAME
```

there are no ways of handling the array as a whole and there are no mathematical functions to allow you to perform matrix manipulation.

An array may be passed as an argument to an external subroutine just like any other variable (R83 Pick programmers should note that the MAT keyword is not required), and within the subroutine the array can be declared without a size specification, like this:

```
SUBROUTINE SUB001(LOCALARR, LOCALB, LOCALC)
DIMENSION LOCALARR()
```

provided that it is declared explicitly in the calling program. An array which is declared without a size specification in a subroutine may be resolved by a subsequent MATREAD statement:

```
MATREAD RECORD FROM ...
```

Dimensioned arrays which are defined within the COMMON area must be declared with an explicit numeric value for the size.

Logical expressions involving dimensioned arrays may use a **wild card** specification in references such as:

```
IF INCOME(*) < 0 ELSE CRT 'All elements positive'
IF SCORE(*,*) = 0 THEN CRT 'There is a zero'
IF SCORE(3,*) > 99 THEN CRT 'Value in row 3 too big'
IF SCORE(*,5) < 0 THEN CRT 'Negative in last col'
```

and will test all the elements of the array – or all those in a specific row or column – and return a value of *true if any one (or more) of the elements of the array satisfy the test*. The same forms are available with *dynamic arrays*, as we see later.

You may assign names to the individual elements of a dimensioned array:

```
EQUATE NAME TO RECORD(1)
EQUATE DATE TO RECORD(7)
```

and this is particularly useful when you are writing code to handle file data which has been read in by the MATREAD statement.

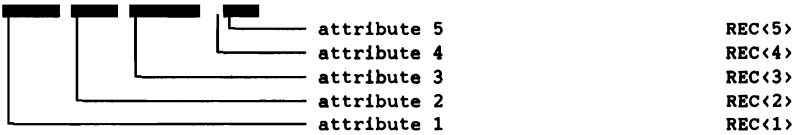
16.4 Dynamic arrays

Advanced Pick Basic supports a most useful data structure known as a **dynamic array**. A dynamic array is simply a **string** which contains, amongst other characters, the system delimiters:

- o The **attribute-mark**, ASCII character 254, and/or

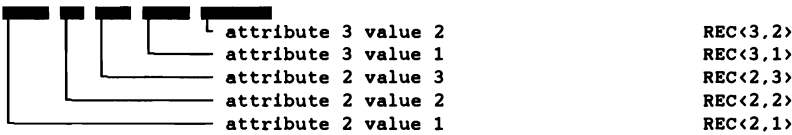
- The **value-mark**, ASCII character 253, and/or
- The **subvalue-mark**, ASCII character 252.

JONES^1234^46.222^^YES



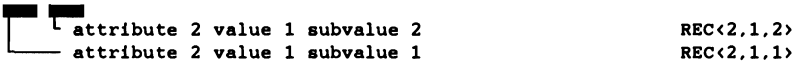
(a) Attributes only

JONES^1000]34]200^20.1]25.122^^YES



(b) Attributes and values

JONES^300\700]34]200^20.1]25.122^^YES



(c) Attributes, values and subvalues

Figure 16.8: Dynamic arrays

You have probably realised that the structure of a dynamic array is identical to that of a normal Pick item as it is held on a file. A set of special functions allows you to manipulate dynamic arrays. You can:

- Pick out the contents of individual elements – attributes, values or subvalues – from the array.
- Insert new elements at the front of the array, at the end of the array, or elsewhere within the array.
- Remove elements from the array.

Some examples of dynamic arrays can be seen in Figure 16.8. The dynamic array in (a) consists of a sequence of **attributes**; that in (b) shows how the **values** of the same dynamic array might look if some of the attributes were **multivalued**; that in (c) shows the appearance if some of the values had **subvalues**; subvalues are rarely encountered in practice. The references to the right of each line show how that element of the array would be identified in Basic assignment statements such as:

```
NAME = REC<1>  
TOTAL = REC<2> * REC<3>  
REC<5> = 'NO'  
REC<4> = NEW.VALUE  
PRINT REC<2,1,1>  
REC<2,1,2> = ''
```


It is this notation and usage which earns them the name **dynamic array**. The adjective **dynamic** arises from the fact that (unlike a dimensioned array) a dynamic array is not fixed in size and elements may be inserted, appended and deleted from the structure.

In Figure 16.8, I have used the ^ character to represent the **attribute-mark** which separates the individual attributes of the array, the] character to represent the **value-mark** which separates the individual values, and the \ character to represent the **subvalue-mark** which separates the individual subvalues. Note that attribute 4 holds a null data value and is represented by just the following attribute-mark. Any references to REC<4> or to REC<6> onwards are acceptable and would refer to null values.

The individual elements of a dynamic array are identified by references of the form:

```
name {<attribute {,value {,subvalue} } >}
```

For example, let us imagine that a variable called DYNARR holds the following dynamic array:

```
AA^B]BB]BBB^C]c\cc\ccc]CC^DD
```

then the following statements:

```
PRINT DYNARR
```

would display the entire contents of the variable (including any attribute-marks, value-marks and subvalue-marks):

```
AA^B]BB]BBB^C]c\cc\ccc]CC^DD
```

```
PRINT DYNARR<1>
```

would display the contents of attribute 1 of the dynamic array held in DYNARR (including any value-marks and subvalue-marks):

```
AA
```

```
PRINT DYNARR<3>
```

would display the contents of attribute 3 of the dynamic array held in DYNARR (including any value-marks and subvalue-marks):

```
C]c\cc\ccc]CC
```

```
PRINT DYNARR<3,2>
```

would display the contents of value 2 of attribute 3 of DYNARR (including any subvalue-marks):

```
c\cc\ccc
```

```
PRINT DYNARR<3,2,1>
```

would display the contents of subvalue 1 of value 2 of attribute 3 of DYNARR:

c

If we were to apply each of the following statements one after the other:

DYNARR<1> = 'QQQ'

would replace the contents of attribute 1 of DYNARR (and any value-marks and subvalue-marks) by the string QQQ:

QQQ^B]BB]BBB^C]c\cc\ccc]CC^DD

DYNARR<9> = 'XX'

would replace the contents of attribute 9 by the string XX. If there were fewer than 9 attributes in the array, sufficient attribute-marks would be added to position XX at attribute 9:

QQQ^B]BB]BBB^C]c\cc\ccc]CC^DD^^^^XX

DYNARR<0> = 'YES'

would insert a new attribute YES at the front of the data in DYNARR, moving all the existing data up by one attribute:

YES^QQQ^B]BB]BBB^C]c\cc\ccc]CC^DD^^^^XX

DYNARR<3,2> = 'bbb'

would replace the contents of value 2 of attribute 3 of DYNARR (and any subvalue-marks) by the string bbb:

YES^QQQ^B]bbb]BBB^C]c\cc\ccc]CC^DD^^^^XX

DYNARR<4,2,1> = ''

would replace the contents of subvalue 1 of value 2 of attribute 4 of DYNARR by null:

YES^QQQ^B]bbb]BBB^C]\cc\ccc]CC^DD^^^^XX

DYNARR<-1> = 'QW'

would **append** the string QW as a new attribute to the end of DYNARR:

YES^QQQ^B]bbb]BBB^C]\cc\ccc]CC^DD^^^^XX^QW

DYNARR<1,-1> = 'A123'

would append the string A123 as a new value to the end of attribute 1 of DYNARR:

YES]A123^QQQ^B]bbb]BBB^C]\cc\ccc]CC^DD^^^^XX^QW

Note that these statements add the necessary delimiter characters.

It is possible for the elements of a *dimensioned* array to be *dynamic* arrays; in this case, these may be manipulated by extending this notation:

DIMARR(10)<7>

DIMARR(5)<1,2>

DIMARR(9)<3,7,1>

```
DIMARR2(4,5)<3>
```

and so on. If we add the substring notation to this, we get terrifying forms such as:

```
DIMARR(10)<7>[1,1]
DIMARR2(3,4)<5,6,7>[8,9]
```

The following points about dynamic arrays are important:

- A dynamic array is held in an ordinary variable and is stored and handled exactly like a *string*, and, like ordinary variables, a dynamic array is not declared explicitly. Only the way in which it is *used* determines whether or not it is to be regarded as a dynamic array.
- If required, a dynamic array may be initialised by a statement such as:

```
DYNARR = ''
```

- A dynamic array may have any number of attributes, values and/or subvalues up to the maximum length for a string.
- Elements may be inserted, changed and/or removed from a dynamic array. The standard functions for handling dynamic arrays add or remove the necessary delimiter characters.
- When a piece of data is to be located or retrieved from a dynamic array, the run-time processor scans the entire string counting the system delimiters. Thus, if we use a reference such as:

```
DATE = REC<7>
```

the variable REC will be scanned, character by character, looking for the sixth and the seventh attribute-marks; the higher the subscript, the longer the search. If you need to use the same attribute several times in a program, it is more efficient to assign it to a simple variable, as here, and then use this variable DATE instead of further references to REC<7>.

In general, array references such as REC<1> or REC<1,2,3> can be used in exactly the same context as ordinary variables, except that the compiler will reject the forms:

```
INPUT REC<1>
INPUT REC<1,2,3>
EQUATE NAME TO REC<1>
```

There are two statements for transferring data between a dimensioned array and a dynamic array:

```
MATBUILD STRING FROM INCOME
STRING = INCOME
```

either of these will copy all the elements from the dimensioned array INCOME into the variable STRING, separating the elements by the attribute-mark.

```
MATPARSE INCOME FROM STRING
INCOME = STRING
```

either of these will use the fields within the variable **STRING** separated by attribute-marks and load these as successive elements into the dimensioned array **INCOME**.

16.4.1 Which type of array to use

Both dimensioned arrays and dynamic arrays can be used for building and holding lists of data used by the program and for holding data which is read from disk files and/or which is to be written to disk files. In these situations, the programmer is often faced with the decision of whether to use a *dimensioned* array or a *dynamic* array.

The following points may influence your decision:

- When you are using dimensioned arrays to process very large records – such as text documents – which may have a large number of attributes, you may be hindered by the maximum number of program variables (including those variables held in **COMMON** and elements of all your dimensioned arrays) which can be used.
- Each dynamic array is a simple string variable and may have any number of elements and may be of any length up to the maximum permitted length of a string.
- A dimensioned array may have no more than two subscripts. The three subscripts available with a dynamic array can be used to simulate a three-dimensional array. If you need more than three dimensions to an array, then you must simulate this yourself.
- When putting data into and taking data from an array, it is quicker to access the elements of a dimensioned array than it would be if the data were held as attributes of a dynamic array. This is because the start address of each element of a dimensioned array can be calculated directly from the subscript (since each element is of the same length); whereas, in order to find a particular attribute within a dynamic array, the run-time processor has to scan the entire string looking for the required attribute marks. The time difference typically involves a factor of 10 or more. For example, if you had statement:

```
DYNARR<200>=DYNARR<200>+1
```

the run-time processor would (twice) have to scan every character in the **DYNARR** string up to the 201st attribute-mark to find the required attribute. In contrast, the locations of the elements in a statement such as:

```
DIMARR(200) = DIMARR(201)+1
```

can be calculated directly by the compiler or the run-time processor.

- The number of elements in a dimensioned array is static. It requires detailed coding to insert new values *within* a dimensioned array (shifting all the following elements up one position) and/or to remove values *from* a dimensioned array (shifting all the following elements down one position). In contrast, you may insert, append and/or remove an unlimited number of attributes to a dynamic array. This is

conveniently exploited when a dynamic array is used to build and maintain a list of data for use in a program.

- There are standard facilities to locate, add up and sort the elements of a dynamic array. There are no standard facilities for finding a specific value within a dimensioned array.
- MATREAD and MATWRITE statements must handle the raw data from the disk (which is physically held as a dynamic array) and use this to load the elements of a dimensioned array. For this reason, these statements are slower than the corresponding READ and WRITE statements which handle dynamic arrays directly as they are read from disk.
- The use of the EQUATE statement to name each individual element of a dimensioned array makes it much easier to read and understand the program coding than with a dynamic array reference. Thus, in conjunction with the declarations:

```
DIMENSION DIMARR(20)
EQUATE NAME TO DIMARR(1), DATE TO DIMARR(7)
```

statements such as:

```
PRINT NAME
DATE = DATE + 7
```

are more intelligible and much easier to write and maintain than are statements such as:

```
PRINT DIMARR(7)
DYNARR(7) = DYNARR(7) + 7
```

and:

```
PRINT DYNARR<7>
DYNARR<7> = DYNARR<7> + 7
```

The statements:

```
EQUATE NAME TO DYNARR<1>
INPUT DYNARR<3>
```

cannot be used and would be rejected by the compiler.

As you see, there are no hard and fast rules ... just points to be considered.

16.5 Program structures

As with most programming languages, the statements fall into four distinct groups:

- Those which are executed **sequentially**:

```
PRICE = COST * (100+RATE)
PRINT PRICE
```

and pass control on to the next physical statement in the program.

- **Selection:** those which test the data and take one of several different courses of action according to the result:

```
IF ANSWER = 'Y' THEN GOTO 1000 ELSE GOTO 2000
CASE CODE='A'; GOSUB Add.routine
ON CODE GOSUB 100,200,300,400
```

- **Iteration:** those which control the number of times which a group of statements is repeatedly executed:

```
FOR NUMBER = 1 TO 10
  TOTAL = TOTAL + NUMBER
  PRINT NUMBER, TOTAL
NEXT NUMBER
```

- Those which **pass control** to and from another piece of program code:

```
GOTO 100
GOTO 1234
GOTO End.of.job
GOSUB 1000
GOSUB Display.title
CALL MENU.ROUTINE
RETURN
```

As these last examples indicate, there are facilities for passing control to and from **internal subroutines**, and also for creating and calling **external subroutines** which are written as separate coding units.

16.5.1 Statement labels

Before we look at the ways in which Basic presents these facilities, we should say something about **statement labels**. The labels which are used as the destination of statements such as:

```
GOSUB Vet.dataname
GOTO End.of.job
ON OPTION.NUMBER GOSUB Add, Change, Delete
GOTO 100
GOSUB 200
GOTO 200.99
```

may be numeric, numeric with a full stop, or alphanumeric. All labels must be unique, and alphanumeric labels must obey the same rules as variable names. As the alphanumeric labels in the program in Figure 16.1 illustrate:

- They must begin with a letter,
- They may consist of any letters, digits, full stops and/or dollar signs. Later releases allow the underscore character `_` to be used in statement labels and variable names.
- They must be followed by a colon. The use of the colon in the

statements which jump to a label is optional:

```
GOTO End.of.job
GOTO End.of.job:
GOSUB Display.title
GOSUB Display.title:
```

Statement labels need only be used on the statements which are the target of GOSUB and GOTO statements. The use of labels on other statements is optional, and this can be an aid to documenting the program.

16.5.2 IF structure

Selection statements perform a specific action according to the result of a test carried out on the data. There are several such statements in Basic:

- o The IF statement,
- o The CASE statement, and
- o The computed GOTO, and
- o The computed GOSUB statements.

The IF statement may take any of the forms shown in Figure 16.9. When the statement is executed, the *condition* (A=B) will be tested; if the result is *true* (if A is equal to B), then the statement(s) in the THEN clause (A=0) will be executed; if the result is *false* (if A is not equal to B), then the statement(s) in the ELSE clause will be executed. The IF statement may be a single-line statement, as in example 1, or it may be presented as a multiline structure, like example 4. In general, the coding is easier to write, to read and to understand if it is written as a multiline structure with the dependent statements (A=0 and B=0 in these examples) indented, as illustrated.

1 IF A=B THEN A=0 ELSE B=0	6 IF A=B THEN A=0 ELSE B=0 END
2 IF A=B THEN A=0	7 IF A=B THEN A=0 END
3 IF A=B ELSE B=0	8 IF A=B ELSE B=0 END
4 IF A=B THEN A=0 END ELSE B=0 END	9 READ REC FROM ID THEN A=0 END ELSE B=0 END
5 IF A=B THEN A=0 END ELSE B=0	

Figure 16.9: IF statement

In any of these forms, the condition $A=B$ can be any *logical expression* and the statements $A=0$ and $B=0$ may themselves be any statement, any sequence of statements, or any other structure. Here are some more examples:

```
IF ANSWER='YES' THEN STOP
```

terminate the program execution if the variable ANSWER contains the string YES.

```
IF TOTAL > MAXIMUM THEN CRT 'TOO BIG'
```

display the message TOO BIG if TOTAL contains a value greater than that in MAXIMUM.

Two or more test may be linked by the AND and OR connectives:

```
IF A>10 AND C<10 THEN STOP
```

terminate the program execution if A contains a value greater than 10 and C contains a value less than 10.

```
IF A<0 OR B<0 THEN GOTO 100
```

jump to statement label 100 if A contains a value less than 0 or B contains a value less than 0, or both.

There are several other Basic statements which have the same THEN/ELSE structure, including:

```
OPEN ...
READ ...
READV ...
MATREAD ...
READNEXT ...
LOCATE ...
INPUT ...
```

and any of these may be presented as a single-line statement or as a multiline structure, as illustrated by example 9 in Figure 16.9.

16.5.3 Logical expressions

The test such as that carried out in the IF statement is sometimes called a **logical expression**. A logical expression is any combination of:

- Variables,
- Literals,
- The logical operators (as illustrated below),
- The logical connectives (AND / OR / & / ! as illustrated below), and/or
- Logical functions (NOT / NUM / ALPHA / ASSIGNED),

which produces a value of *false* or *true*, and they are encountered in several contexts in Basic:

```
IF logical.expression THEN ... ELSE ...
CASE logical.expression
```



```

LOOP ... UNTIL logical.expression DO ...
LOOP ... WHILE logical.expression DO ...
FOR ... UNTIL logical.expression
FOR ... WHILE logical.expression

```

Some general examples of logical expressions in the IF statement are:

```

IF A = B THEN ... ELSE ...
IF A EQ B THEN ... ELSE ...
    the result is true if A is equal to B

```

```

IF A # B THEN ... ELSE ...
IF A NE B THEN ... ELSE ...
    the result is true if A is not equal to B

```

```

IF A < B THEN ... ELSE ...
IF A LT B THEN ... ELSE ...
    the result is true if A is less than B

```

```

IF A > B THEN ... ELSE ...
IF A GT B THEN ... ELSE ...
    the result is true if A is greater than B

```

```

IF A >= B THEN ... ELSE ...
IF A GE B THEN ... ELSE ...
    the result is true if A is greater than or equal to B

```

```

IF A <= B THEN ... ELSE ...
IF A LE B THEN ... ELSE ...
    the result is true if A is less than or equal to B

```

```

IF A MATCH '3N' THEN ... ELSE ...
IF A MATCHES '3N' THEN ... ELSE ...
    the result is true if A is a string of 3 digits

```

```

IF A=0 AND B=0 THEN ... ELSE ...
IF A=0 & B=0 THEN ... ELSE ...
    the result is true if both A=0 and B=0 (false if either test is false).

```

```

IF A=0 OR B=0 THEN ... ELSE ...
IF A=0 ! B=0 THEN ... ELSE ...
    the result is true if A=0 and/or B=0 (false if both tests are false).

```

In those statements which require the THEN/ELSE clauses, the statement(s) in the THEN clause will be executed if the result is true, and the statement(s) in the ELSE clause will be executed if the result is false.

In statements such as OPEN, READ, LOCATE, the statement(s) in the THEN clause will be executed if the operation is successful, and the statement(s) in the ELSE clause will be executed if the operation is unsuccessful.

16.5.4 Pattern matching

Pattern matching offers a means of checking the format of a string.

Look at these examples:

```
IF CODE MATCHES '3N' THEN ...
```

would be *true* if CODE contained three numeric digits. So this would be *true* for values such as 111 and 000 and 999 and 098, but it would be *false* for values such as 11 and 1234 and AAA and 1A2.

```
IF CODE MATCHES '3N/2A' THEN ...
```

```
IF CODE MATCHES "3N/'2A" THEN ...
```

would be *true* if CODE contained three numeric digits followed by the character / followed by two letters. So it would be *true* for values such as 111/AA and 000/AM and 999/PG and 098/ab, but not for values such as 11/A and 111/a and 11/AA and 99/PGA.

The general form of pattern matching expressions is:

string MATCH pattern

or:

string MATCHES pattern

The *pattern* is an expression made up of any combination of the elements:

nN checks that the value comprises only *n* numeric digits, that is, 0 to 9.

nA checks that the value comprises only *n* letters of the alphabet, that is, A to Z (upper-case) and a to z (lower-case).

nX checks that the value comprises exactly *n* characters of any type, including <Ctrl> keys, the <Esc> key, attribute-marks, and so on. In effect, this checks just the length of the string.

ON checks that the entire value comprises only numeric digits, irrespective of length.

OA checks that the entire value comprises only letters of the alphabet, irrespective of length.

'xx' checks for the string *xx*. If it is likely to be ambiguous or confused with the above patterns, such a string may be enclosed in apostrophes, quotation marks or backslash characters, otherwise these may be omitted.

To check that a value *does not* match a certain pattern, you will use forms such as:

```
IF CODE MATCH '3N' ELSE PRINT 'Not 3 digits'
IF NOT(CODE MATCH '3N') THEN PRINT 'Not 3 digits'
CASE NOT(CODE MATCH '3N')
```

16.5.5 CASE structure

The IF statement tests the value of a logical expression and takes one of *two possible courses of action*. The CASE statement offers a means of taken any one of *several courses of action*, according to a sequence of tests. The structure of the CASE statement is illustrated by the examples in Figure 16.10.

Let's start by looking at fragment (a). As this is executed, the first CASE condition ($A=1$) will be tested: if this is true then the statements:

```
B=A
C=A
```

will be executed, and the processing will then jump out of the structure – to statement number 99 in this fragment. If the CASE condition $A=1$ is not true, then the next CASE condition ($A=2$) will be tested: if this is true, then the statements:

```
B=B*2
C=C*2
```

will be executed, and the processing will then jump out of the structure – to statement number 99. This testing will continue until all the CASE conditions have been tested (and you may have as many CASE sections as you wish): if none of the conditions is true, then the processing will *fall straight through* the CASE structure and carry on with the processing following the END CASE statement (statement number 99).

```
BEGIN CASE
  CASE A=1
    B=A
    C=A
  CASE A=2
    B=B*2
    C=C*2
  CASE A=0 OR B=0
    C=0
  END CASE
99 * Rest of program
```

(a)

```
CRT 'Enter option N / C / D / S: ':
INPUT REPLY
BEGIN CASE
  CASE REPLY='N'; ALL NEW.ROUTINE
  CASE REPLY='C'; CALL CHANGE.ROUTINE
  CASE REPLY='D'; CALL DELETE.ROUTINE
  CASE REPLY='S'; STOP
CASE 1
```

```
CRT 'Enter N)ew/C)hange/D)elele/S)top'
```

```
END CASE
```

(b)

Figure 16.10: CASE structure

Look at fragment (b) in Figure 16.10. What happens if the user's response isn't any of the expected answers? What happens if the user's REPLY is, say, YES instead of the N, C, D, or S which we are expecting? In most situations, you will want to take some action if *none* of the CASE tests is satisfied. What we need is some condition which will *always* be true, even if the other CASE statements fail. The conventional way of doing this is a sequence such as:

```
CASE 1
```

```
CRT 'Re-enter N)ew / C)hange / D)elele / S)top'
```

The result of *CASE 1* is *always true* (recall that a value of 0 is false and any other numeric value is true).

Some points to note about the CASE structure:

- o The entire set of CASE statements is enclosed in a pair of BEGIN CASE / END CASE statements.
- o There may be any number of CASE statements within the structure.
- o The statements which are dependent upon a CASE statement may be written on the same line (separated by a semi-colon), or as separate lines.
- o The first CASE statement which is satisfied is acted upon and the processing leaves the structure, ignoring any later CASE statements.
- o Each CASE statement is quite independent of the others.
- o One CASE structure may be nested within another.

In practice, you will find the CASE structure a very convenient coding device, and in many situations more manageable and easier to write and extend than the equivalent IF structure(s).

16.5.6 ON ... GOTO / ON ... GOSUB statements

Two further statements simulate the multiple testing that you get with the CASE structure: ON ... GOTO and ON ... GOSUB. These are sometimes known as the **computed GOTO statement** and the **computed GOSUB statement**, as illustrated in Figure 16.11.

Look at the ON ... GOTO statement shown here. When this is executed, the value held in CODE will be tested: if the value is 1, then the processing will jump to statement label 100, if the value is 2, then the processing will jump to statement label 177, and so on. If the value of

CODE is less than 1 or (in this case) greater than 6, then no action is taken and the processing passes on to the next statement.

```
ON CODE GOTO 100, 177, 300, 445, 177, 345
ON CODE GO 100, 177, 300, 445, 177, 345
-----
ON VALUE+1 GOSUB Reject,200,Final,200,300,300,Final
```

Figure 16.11: ON ... GOTO / ON ... GOSUB

The ON ... GOSUB statement has a similar effect, jumping to an *internal subroutine* and subsequently returning to the statement immediately following the ON ... GOSUB statement.

16.5.7 Iteration

There are two structures which can be used to control the number of times which a piece of processing is executed: the FOR ... NEXT structure, and the LOOP structure.

16.5.8 FOR / NEXT structure

As the examples of the FOR / NEXT structure in Figure 16.12 illustrate, you will mark the *start* of the section to be repeated by the FOR statement:

```
FOR variable = first TO last {STEP incr} {UNTIL condition}
or:
FOR variable = first TO last {STEP incr} {WHILE condition}
```

and the *end* of the repeated section is marked by the NEXT statement:

```
NEXT variable
```

where *variable* is a variable which is to be set and incremented at each iteration; *first* and *last* are the limits over which variable is to be incremented. The variable is set to *first* when the structure is first executed, incremented by *incr* at each iteration, and the iterations continue until:

- The control *variable* reaches the value *last* or ...
- The *condition* is satisfied (that is, when the UNTIL condition is reached, or when the WHILE condition is no longer true).

The examples in Figure 16.12 illustrate these points.

```
FOR COUNTER = 1 TO 100
    TOTAL = TOTAL + RECORD<COUNTER>
    PRINT RECORD<COUNTER>
NEXT COUNTER
-----
```

```
FOR COUNTER = 1 TO 100 STEP 2
  TOTAL = TOTAL + COUNTER
NEXT COUNTER
-----
FOR COUNTER = 1 TO COUNTER+1 WHILE TOTAL<= 1000
  TOTAL = TOTAL + RECORD<COUNTER>
  PRINT RECORD<COUNTER>
NEXT COUNTER
-----
FOR CTR = 1 TO CTR+1 UNTIL RECORD<CTR>=''
  TOTAL = TOTAL + RECORD<CTR>
  PRINT RECORD<CTR>
NEXT CTR
```

Figure 16.12: FOR /NEXT structure

16.5.9 LOOP structure

The LOOP structure is a very powerful means of controlling iteration and has one of the forms shown in Figure 16.13. Either of the two blocks of statements may be omitted, and the UNTIL / WHILE ... DO element may be omitted if an indefinite loop is required. Programmers who have worked with languages such as C and Pascal will recognise that the LOOP structure performs the roles of several iteration statements in other languages.

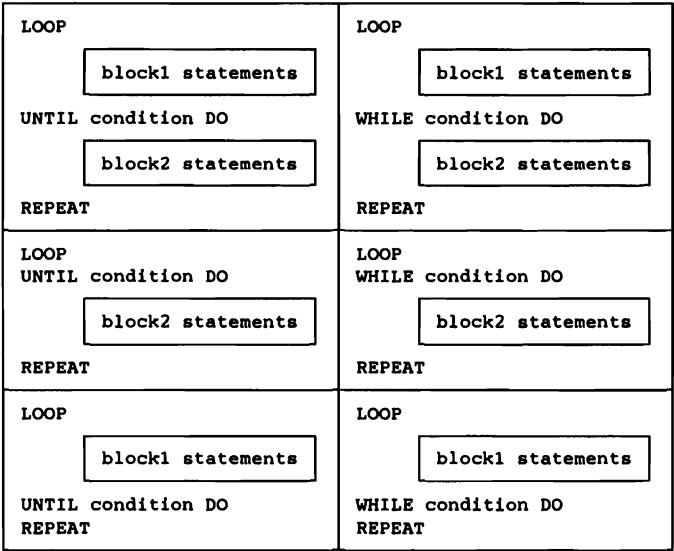


Figure 16.13: LOOP structure

```
LOOP
  PRINT MSG1;; INPUT CODE
UNTIL CODE='.' DO
```

```

    GOSUB 100
  REPEAT
  -----
  LOOP UNTIL STRING='' DO
    PRINT STRING[1,1]
    STRING=STRING[2,LEN(STRING)]
  REPEAT
  -----
  LOOP
    CRT 'Enter Y or N: ':
    INPUT RESPONSE
  UNTIL RESPONSE='Y' OR RESPONSE='N' DO REPEAT
  -----
  LOOP
    CRT 'Enter Y or N: ':
    INPUT RESP,1
  UNTIL RESP NE '' AND INDEX('YyNn',RESP,1) DO
    CRT 'Please enter Y for YES or N for NO'
  REPEAT
  -----
  LOOP
    PRINT MSG2:; INPUT CODE
  WHILE TRIM(CODE) = '' DO REPEAT
  -----
  LOOP
  * Do this indefinitely
    GOSUB Get.data
    GOSUB Analyse.data
  REPEAT

```

Figure 16.14: Some LOOP structures

When the LOOP statement is processed ...

- 1) any statements in *block1* are executed, then ...
- 2) the UNTIL / WHILE *condition* is tested. If the condition is satisfied (that is, if the UNTIL condition is true, or if the WHILE condition is no longer true), then the loop is terminated; if the *condition* is not yet satisfied, then ...
- 3) any statements in *block2* are executed then the loop starts again from the beginning.

Some examples are shown in Figure 16.14.

The EXIT statement allows the processing to leave an iteration fairly elegantly by jumping outside the nearest NEXT or REPEAT statement.

16.5.10 B/LIST: indenting the source code

When you are using multiline structures such as IF, CASE, FOR and LOOP, it is easy to forget an END, END CASE, NEXT or REPEAT statement, and it is often difficult to determine where a missing statement should go. The program indentation utility **B/LIST** will indent the source

statements of your program, thereby making it easier to detect faulty structures in your coding. B/LIST is invoked by a TCL command of the form:

```
B/LIST file.name {item.list} {(options)}
```

For example:

```
B/LIST WAGES.BP * (P
B/LIST MBPROGS SENTENCE01 (FO
```

The utility will display or print a listing showing the logical indentation, and there are options to indent the statements of the source program. Amongst the most useful options are:

- F save the formatted program on the file and suppress the display.
- o overlay the original program. Otherwise, the formatted program is saved as xxx.BLIST.
- P send the listing to the printer.
- R force all indentations to zero. This is useful for **compressing** source programs in order to save space.
- S draw vertical lines on the listing to indicate the program structure.

EPick gives full details of the utility and the options which are available.

16.6 Calculations and assignment statements

Calculations are carried out by means of the **Basic assignment statement**. This has forms such as:

```
RATE = 23
RESPONSE = "YES"
COST = PRICE * VAT.RATE
LET RATE = 23
LET COST = PRICE * VAT.RATE
```

The keyword LET is optional. In all cases, the **expression** to the right of the = sign is evaluated, and the result assigned to the variable (or array element) specified to the left of the = sign.

There are three main types of assignment statement:

- o Arithmetic assignments,
- o String assignments, and
- o Logical assignments.

Arithmetic assignments use *arithmetic expressions*, and these are made up of numeric values, literals, arithmetic functions and arithmetic operators:

```
ANSWER = 42
ROOT = (0 - B - SQRT(B * B - 4 * A * C)) / (2*A)
```


The *arithmetic operators* are shown in Figure 16.15 (a).

String assignments use alphanumeric *string expressions*, and these are made up of string values, literals, string functions and string operators:

```
RESPONSE = "YES"
INITIAL = FIRST.NAME[1,1]
SALUTE = 'MRS ' : INITIAL : ' ' : SURNAME : ', '
UNDERLINE = STR(' ',LEN(SALUTE))
```

The *string operators* are shown in Figure 16.15 (b). Substrings may be extracted from a string by the *square-bracket* notation:

```
SURNAME = 'ABCDEFGH'
KEYX = SURNAME[2,3]
```

which will *extract* three characters from the string in SURNAME, starting at the second character, and assign the result (BCD in this case) to KEYX. The same notation can be used to *replace a substring* within a string:

```
STRING = 'THE CAT SAT ON THE MAT'
STRING[5,3] = 'ABCDE'
```

would insert the string ABCDE into the contents of STRING, replacing the three characters starting at the fifth character, leaving THE ABCDE SAT ON THE MAT in STRING.

Logical assignments use *logical expressions*, and these are made up of logical values, literals, logical functions and logical operators and connectives:

```
MPENSIONER = AGE>65 AND SEX='M'
PENSIONER = MPENSIONER OR FPENSIONER
ELIGIBLE = NOT(PENSIONER)
```

The *logical operators* and *connectives* are shown in Figure 16.15 (c).

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
\	integer division, returning the remainder
** ^	exponentiation

(a) Arithmetic operators

Operator	Meaning
:	concatenation

CAT	concatenation
[a,b]	substring extraction / replacement

(b) String operators

Operators	Meaning
= EQ	equal
# NE <> ><	not equal
< LT	less than
> GT	greater than
<= LE	less than or equal to
>= GE	greater than or equal to
AND &	and
OR !	or

(c) Logical operators and connectives

Figure 16.15: Basic assignment operators

16.6.1 Operator assignment

The feature known as **operator assignment** allows you to write expressions such as:

A += B

instead of the more usual:

A = A + B

Statement	Equivalent
AAA += BBB	AAA = AAA + BBB
AAA -= BBB	AAA = AAA - BBB
AAA := BBB	AAA = AAA : BBB
AAA *= BBB	AAA = AAA * BBB
AAA /= BBB	AAA = AAA / BBB
AAA \= BBB	AAA = AAA \ BBB

Fig 16.16: Operator assignment

This can be used with other operators, as shown in Figure 16.16.

16.6.2 Initialising variables / CLEAR

All variables (and elements of arrays) should be initialised, that is, assigned a value, before they are used as the source of data in an expression. Thus, if the first statement in a program were:

VALUE1 = VALUE2 + 333

and no value had yet been assigned to VALUE2, the program would report a non-fatal execution error:

Variable has not been assigned a value; zero used!

and the value 333 would be assigned to VALUE1, leaving VALUE2 unassigned. Typically, variables are initialised by statements of the form:

```
VALUE2 = 0
STRING = ''
```

or the CLEAR statement may be used to assign 0 to all the variables and dimensioned array elements in the program. Typically, the CLEAR statement will be used at the start of the program. If you use it elsewhere, you should bear in mind that:

- In the main program, CLEAR will clear the contents of all COMMON variables and all local variables, but will not clear labelled COMMON variables.
- In an external subroutine, CLEAR will only clear the contents of local variables.

The ASSIGNED function is available to detect whether a variable has been assigned a value:

```
IF ASSIGNED(VALUE3) ELSE VALUE3 = 0
IF NOT(ASSIGNED(VALUE3)) THEN VALUE3 = 0
```

and might be used to check the state of data passed to an external subroutine.

16.6.3 Basic functions

There are a number of standard **functions** – often called the **intrinsic functions** – available for use with Basic. These perform standard processing using data supplied by the program. For example, the SQRT function used in the statement:

```
ROOTX = SQRT(RESULT)
```

will take the value held in the variable RESULT – this value is known as the **argument** for the function – calculate the square root of this number and pass the result back to the program where it is assigned to ROOTX. Some functions have more than one argument:

```
SPARE = MOD(AVAILABLE, BOXSIZE)
```

divide the contents of AVAILABLE by the contents of BOXSIZE and return the remainder, storing it in SPARE in this example..

```
WORD5 = FIELD(SENTENCE, ' ', 5)
```

scan the string held in SENTENCE and return the substring enclosed by the fourth and fifth spaces, that is, the fifth word, placing this in WORD5.

Some functions have no arguments, but the parentheses must be used:

```
TODAY = DATE()
NOW = TIME()
```

return the current date and time (in their internal formats).

It is possible to have functions within functions:

```
INITIAL = OCONV(TRIM(NAME), 'MCU')
EXTERNAL.DATE = OCONV(DATE(), 'D2/')
FOURTH.ROOT = SQRT(SQRT(NUMBER))
```

These are similar to the functions encountered in other languages, although, unlike some languages, you do not need to include any special routines in your source program in order to be able to use these standard functions, the necessary routines are automatically incorporated as and when required. It is not possible to define your own functions in Basic on Advanced Pick.

Let us look at some of the more useful functions.

- Ⓔ is used for cursor-positioning and producing special screen/graphic effects, as we see in more detail later.

```
PRINT @(0,23):'BOTTOM OF PAGE'
PRINT @(-1):
```

CHAR generates an ASCII character by specifying the integer value of the character. This is complementary to the **SEQ** function. A chart of ASCII characters appears in the Appendix.

```
EQUATE AM TO CHAR(254); ** Attribute mark
PRINT CHAR(7):'Invalid data'
```

COUNT returns the number of occurrences of a given substring within a string.

```
WORDS = COUNT(SENTENCE,' ') + 1
```

DATE() returns today's date in the internal form, that is, the number of days since 31/12/67.

```
PRINT 'Today is ':OCONV(DATE(),'D')
AGE = DATE() - AGE.FIELD
```

DCOUNT counts the number of occurrences of a substring within a string, the substrings being separated by any field-separator.

```
WORDS = DCOUNT(SENTENCE,' ')
```

FIELD assumes that a string comprises substrings separated by a field-separator, and then extracts a specific substring.

```
PRINT 'SECOND WORD = ':FIELD(SENTENCE,' ',2)
```

After this function has been used, two further functions **COL1()** and **COL2()** will return the position of the space before and after the second word in the sentence.

FOLD breaks a string into substrings of a specific length separated by specified character(s).

```
PRINT FOLD(TEXT, 20, CHAR(10):CHAR(13))
```

ICONV

oconv these functions apply an Access processing code to a value, as we see later.

```
INTERNAL.DATE=ICONV(EXTERNAL.DATE,'D')
EXTERNAL.DATE=OCONV(INTERNAL.DATE,'D')
UPPER.CASE.NAME = OCONV(NAME,'MCU')
```

INDEX returns the position of the start of a specified occurrence of a substring within a string.

```
CRT 'Second THE starts at ':INDEX(PHRASE,'THE',2)
```

This is also used to detect the presence of a substring within a string:

```
IF INDEX(ANSWER,'YES',1) THEN GOTO Accepted
```

and would be faster than the equivalent:

```
IF COUNT(ANSWER,'YES')>0 THEN GOTO Accepted
```

INT returns the integer part of a number.

```
IF VALUE=INT(VALUE) THEN PRINT 'WHOLE NUMBER'
```

LEN returns the length of a string.

```
PRINT NAME:' is ':LEN(NAME):' characters long'
```

MOD

REM either of these will return the remainder when the first operand is divided by the second.

```
LEFT.OVER = MOD(TOTAL, BOXSIZE)
LEFT.OVER = REM(TOTAL, BOXSIZE)
LEFT.OVER = TOTAL \ BOXSIZE
```

NOT reverses the *true* or *false* value of a logical expression.

```
CASE NOT(CODE MATCH '1A')
```

NUM returns a value of *true* if the value is numeric, otherwise returns a value of *false*.

```
IF NUM(VALUE) THEN PRINT 'NUMERIC VALUE'
```

RND returns a **random number**. For example, RND(6) will return a random integer number in the range 0, 1, 2, 3, 4, or 5.

```
WINNER = LIST(RND(SIZE)+1)
```

SEQ returns the decimal value of any ASCII character. This is complementary to the CHAR function.

```
DECIMAL = SEQ(LETTER)
```

SPACE generates a string of spaces of a specific length.

```
PRINT 'A':SPACE(30):'B'
```

SQRT returns the square root of a numeric value.

```
ROOT1 = SQRT(B*B - 4*A*C)
```

STR generates a string of characters repeated a specific number of times.

```
PRINT STR('*',80)
```

SYSTEM returns a range of information about the system and the users, as we discuss later.

TIMEDATE() returns the time and the date in the external form:

```
12:29:59 29 Jul 1995
```

TIME() returns the current time in the internal form, that is, as a number of seconds since midnight.

```
PRINT 'It is now ':OCONV(TIME(),'MTS')
DELAY = TIME() - START.TIME
```

TRIM removes leading spaces and trailing spaces from a string and reduces any sequence of embedded spaces to a single space.

```
INPUT.NAME = TRIM(INPUT.NAME)
```

As the table in the Appendix shows, there are many more functions available; there are also the %xxx functions which interact with the host environment on AP/DOS and AP/Unix systems. EPick gives full details.

16.6.4 ICONV / OCONV functions

Function	Result
ICONV('29.7.99','D')	11533
OCONV(11533,'D')	29 Jul 1999
OCONV(11533,'DWA')	Thursday
OCONV(12345,'MTS')	03:25:45
OCONV('29.7.99','G1.1') *	7
OCONV('MARY JONES','MCT') *	Mary Jones
OCONV('MARY JONES','MCL') *	mary jones
OCONV('mary jones','MCU') *	MARY JONES
ICONV(1234.5678,'MD2')	123457
OCONV(123456789,'MD2,')	1,234,567.89
OCONV(0,'U50BB') *	17 mary dm
OCONV(0,'U60BB') *	mary dm

Figure 16.17: ICONV / OCONV

The Basic ICONV and OCONV functions impose a **processing code** on a data value, typically converting it to some other form. The general form of the functions is:

```
ICONV(value,code)
OCONV(value,code)
```

where *value* is any expression and *code* is one of the processing codes described below. ICONV converts data to an **internal form** and OCONV converts data to an **external form**. Look at these examples:

```
EXTERNAL=OCONV(INTERNAL, 'D')
takes the number held in INTERNAL and convert this to the external
```

date format, placing the result in EXTERNAL.

INTERNAL=ICONV(EXTERNAL, 'D')

takes the date held in EXTERNAL, in a form such as 31/12/95 or 31 December 1995, and convert this to the *internal date format*, placing the result in INTERNAL.

CAP.NAME=OCONV(NAME, 'MCU')

takes the string held in NAME and convert this to *UPPER-CASE*, placing the converted string in CAP.NAME.

The processing codes which can be used in these functions are D / G / MC / MD / MT / Tn,m / Tfile / U. However, the effects of some of these – notably G and Tn,m and Tfile – can be achieved directly by the facilities of Basic. The examples in Figure 16.17 illustrate the use and results of some examples of the ICONV / OCONV function. In those forms which are marked with an asterisk, either ICONV or OCONV may be used with the same effect. We take a closer look at processing codes in Chapter 14.

16.6.5 SYSTEM function

- 1 Returns 1 if a PRINTER ON statement is effective or the program has been executed with the P option, indicating that PRINT output is going to the *spooler*.
- 2 Returns the current page width.
- 3 Returns the current page depth.
- 4 Returns the number of lines remaining on the current page.
- 5 Returns the current page number.
- 6 Returns the current line count.
- 7 Returns the terminal type code.
- 8 Returns the backing storage block size.
- 9 Returns the CPU millisecond count for the current session.

- 10 Returns 1 if there is data waiting in the stack.
- 11 Returns the number of items in an externally generated select-list.
- 12 Returns the system time, since midnight, in milliseconds.
- 13 Returns 1 and causes a break in processing.
- 14 Returns the number of characters currently held in the terminal type-ahead input buffer.
- 15 Returns the alphabetic options specified on the command which invoked the program.

Figure 16.18: SYSTEM function

The SYSTEM function returns any of a number of details about the state of the system, the current environment, or the user's terminal characteristics, according to a single numeric argument. For example:

```
CRT SYSTEM(2)
```

will display the current page width (in characters), and

```
CRT SYSTEM(3)
```

will display the current page depth (in lines) and

```
CRT SYSTEM(100)
```

will display details of the Advanced Pick release number which is being used.

Some other examples are shown in Figure 16.18. EPick gives full details of what is available.

16.6.6 Facilities for use with dynamic arrays

A **dynamic array** is a string whose contents are organised as a sequence of *attributes* and/or *values* and/or *subvalues*. We saw earlier that the individual elements of a dynamic array are identified by references of the form:

```
DYNARR<a>
DYNARR<a,v>
DYNARR<a,v,s>
```


specifying the *attribute*, the *value* and/or the *subvalue* concerned. There are several standard functions for handling dynamic arrays.

To **extract an element** from a dynamic array, you will use the following forms:

```
PRINT DYNARR<7>
    print attribute 7 of the dynamic array DYNARR.
```

```
VALUE=DYNARR<2> * DYNARR<5>
    multiply attribute 2 by attribute 5 and assign the result to the variable
    VALUE.
```

You may find some R83 Pick programs which use the older EXTRACT function for the same purpose:

```
PRINT EXTRACT(DYNARR,7)
VALUE = EXTRACT(DYNARR,2) * EXTRACT(DYNARR,5)
```

The forms using the <a,v,s> notation and the INS and DEL statements, are *shorthand* forms of the older EXTRACT and other functions shown here. The compiler actually replaces these by the original functions.

To **replace an element** within a dynamic array:

```
DYNARR<7>='ABC'
DYNARR = REPLACE(DYNARR,7;'ABC')
    either of these will replace attribute 7 by the value ABC.
```

```
DYNARR<6>=DYNARR<2> * 100
DYNARR = REPLACE(DYNARR,6;EXTRACT(DYNARR,2) * 100)
    replace attribute 6 by 100 times attribute 2.
```

To **append an element to the end** of a dynamic array:

```
DYNARR<-1>='123'
DYNARR = REPLACE(DYNARR,-1;'123')
DYNARR = INSERT(DYNARR,-1;'123')
    append the value 123 as a new attribute to the end of DYNARR.
```

```
DYNARR<5,-1>='B'
    append a new value to the end of attribute 5 of DYNARR.
```

```
DYNARR<6,7,-1>='C'
    append a new subvalue to the end of value 7 of attribute 6 of
    DYNARR.
```

To **insert a new element** into a dynamic array:

```
INS 'A' BEFORE DYNARR<7>
DYNARR = INSERT(DYNARR,7; 'A')
    insert the value A as a new attribute before attribute 7 of DYNARR.
```

```
INS 'B' BEFORE DYNARR<1>
DYNARR<0> = 'B'
```

```
DYNARR = INSERT(DYNARR,1; 'B')
```

insert the value B as a new attribute before the first attribute of DYNARR.

```
INS 'C' BEFORE DYNARR<5,9>
```

```
DYNARR = INSERT(DYNARR,5,9;'C')
```

insert the value C as a new value before value 9 of attribute 5 of DYNARR.

```
INS 'D' BEFORE DYNARR<1,2,3>
```

```
DYNARR = INSERT(DYNARR,1,2,3;'D')
```

insert the value D as a new subvalue before subvalue 3 of value 2 of attribute 1 of DYNARR.

To delete an element from a dynamic array:

```
DEL DYNARR<7>
```

```
DYNARR = DELETE(DYNARR,7)
```

delete attribute 7 and the associated attribute-mark from DYNARR.

```
DEL DYNARR<5,9>
```

```
DYNARR = DELETE(DYNARR,5,9)
```

delete value 9 of attribute 5 and the associated value-mark from DYNARR.

```
DEL DYNARR<6,7,2>
```

delete subvalue 2 of value 7 of attribute 6 and the associated subvalue-mark from DYNARR.

In all cases, the appropriate element and its associated field separator will be deleted from the dynamic array.

Other statements and functions available for use with dynamic arrays include MATBUILD and MATPARSE which we met earlier and:

sort sort the elements of a dynamic array into ascending, left-justified sequence:

```
SORTED.LIST = SORT(UNSORTED.LIST)
```

sum return the sum of all the attributes (or values or subvalues) of a dynamic array:

```
TOTAL = SUM(DYNARR)
```

16.6.7 LOCATE statement

Of special interest is the LOCATE statement which offers a means of locating a specific element within a dynamic array:

```
LOCATE string IN dynamic.array {BY sequence} SETTING variable  
THEN/ELSE ...
```

The statement shown in Figure 16.19 (a) will scan the dynamic array held

in NAMES.1 and look for the first attribute which consists of the word SMITH. Having found such an attribute, the processor will put the attribute-number of that attribute into WHERE and take the action specified in the THEN clause; the search will then terminate, ignoring any repetitions of SMITH. If it cannot find the word SMITH, then it will take the action specified in the ELSE clause and the variable WHERE will contain a number *one more* than the number of attributes in the dynamic array. Only elements which exactly match the sought string are recognised; the presence of a partial string, such as in the name SHOESMITH, will be ignored.

If the names in the dynamic array are known to be *sorted* in some specific order, say, *Ascending Left-justified*, then the statement may be written as in Figure 16.19 (b). As before, the statement will scan the dynamic array in NAMES.2 looking for the first occurrence of the word SMITH. If the word SMITH can be found, the THEN clause will be executed and the variable POSN will contain the number of the attribute where SMITH was found. However, if there is no SMITH in the array, the search will terminate when a word *alphabetically greater than* SMITH (such as SMYTHE or WATKINS) is found, the ELSE clause will be executed and the value of POSN will contain the position of this attribute, that is the attribute where SMITH *would have been* if it had been in the array and in the correct order.

In all these forms, the THEN clause is executed if the element is found, and the ELSE clause if it is not found. A typical action within the ELSE clause might be to add the value uniquely to the dynamic array. Figure 16.19 (c) shows how we might use this to add names to the list, thus constructing a sorted list of names.

```
LOCATE 'SMITH' IN NAMES.1 SETTING WHERE THEN
  GOSUB 100
END ELSE
  PRINT 'Not found'
END
```

(a)

```
LOCATE 'SMITH' IN NAMES.2 BY 'AL' SETTING POSN THEN
  GOSUB 100
END ELSE
  PRINT 'Not found'
END
```

(b)

```
INPUT NAME
LOCATE NAME IN NAMES.2 BY 'AL' SETTING POSN THEN
  PRINT NAME: ' already in the list'
END ELSE
  INS NAMES.2 BEFORE NAMES.2<POSN>
END
```

(c)

Figure 16.19: LOCATE statement

There is an alternative LOCATE statement:

```
LOCATE(string, dynamic.array; variable {; sequence}) THEN/ELSE ...
```

and the corresponding forms of the above instances would be:

```
LOCATE('SMITH', NAMES.1; WHERE) THEN/ELSE ...
LOCATE('SMITH', NAMES.2; POSN; 'AL') THEN/ELSE ...
```

respectively.

The full forms and facilities of these statements are described in EPick.

16.7 Displayed and printed output

Messages, the results of calculations and other information can be displayed on the screen by means of the PRINT statement or the equivalent CRT statement (incidentally, CRT stands for cathode ray tube, an old name for a screen). Some typical uses of the PRINT and CRT statements are:

```
PRINT "This program uses Simpson's rule"
PRINT PRICE * (100 + VAT.RATE) / 100
CRT 'Please enter your code number: '
```

The **colon** can be used to **concatenate** several values together on the same line:

```
PRINT 'The answer is ': VALUES
CRT "Employee's age is ":AGE:' years'
```

and, if the statement ends in a **colon**, the cursor will be held in position immediately after the message:

```
CRT 'Enter your name: ':
INPUT SURNAME
```

as in this case, waiting for the user's response at that position.

If you want to output several values in *columns*, then you can use **commas**:

```
PRINT A, B, B*2
```

and each individual value will appear at the left of an 18-character wide column. These forms are illustrated by the program and the output in Figure 16.20.

```
* Program to demonstrate the PRINT statement
PROMPT ''
PRINT @(-1):@(25,0):'*** PRINT demonstration ***'
PRINT
PRINT 'What is your name? ':
INPUT NAME
PRINT
```

```

PRINT 'Please enter a number 1 to 50, ':NAME:': ':
INPUT NUMBER
PRINT
PRINT 'Root', 'Number', 'Square', 'Root'
FOR X=1 TO NUMBER
    PRINT SQR(X), X, X*X, X*X*X
NEXT X

```

 ** PRINT demonstration **

What is your name? Gwen

Please enter a number 1 to 50, Gwen: 11

Root	Number	Square	Root
1	1	1	1
1.4142	2	4	8
1.732	3	9	27
2	4	16	64
2.236	5	25	125
2.4494	6	36	216
2.6457	7	49	343
2.8284	8	64	512
3	9	81	729
3.1622	10	100	1000
3.3166	11	121	1331

Figure 16.20: PRINT statement

16.7.1 Format masks

The concept of **masks** allows a format to be imposed on a value as it is output. When a mask is used, the individual elements in the PRINT and CRT statements have the general form:

value {mask}

with the mask immediately following the data value. Some examples might be:

```

PRINT VALUE 'R2'
PRINT VALUE '2'
PRINT KEY 'R%10'
PRINT (NAME ' .....') 'L#10' : NUMBER
PRINT VAL1 MASK1: VAL2 MASK2: VAL3 MASK3

```

A **format mask** comprises a sequence of the following symbols:

```
{j}{s}{.}{n{m}}{xs}{c}
```

where:

- j is the justification (L for left-justification is the default, or R for right-justification), or D for date). If this is omitted, L is assumed.
If the D (date) format is used. the entire mask may be any valid D processing code and the other symbols are redundant.
- s is the dollar sign which is to be placed to the left of the string (the default is no dollar sign).
- . is the comma to be inserted between the thousands, as in Western numbers (the default is no commas).
- n is the number of decimal places to be shown (the default is that of the source data).
- m indicates that the input value is to be **scaled down** by a factor of *m*, that is, the implied decimal point in the source data is shifted *m* places to the left before output.
- x is the fill-character which is to be used to pad the field to the required length, and will be any one of:
 - % the string is to be padded with zeroes (padding to the left on R justification, or to the right on L justification).
 - # the string is to be padded with spaces. This is the default if no fill character is specified.
 - * the string is to be padded with asterisks.
- s is the length of the field. If the source data is too long for the specific mask, it will be truncated at the left (with a right-justified mask) or at the right (with a left-justified mask). If the source data is shorter than this length, then it will be padded (as indicated by the x parameter).
- c Normally, a minus sign is printed immediately before a negative number and no arithmetic sign is shown for a positive number. This single letter code indicates how negative/positive values are to be treated:
 - C append CR to negative numbers.
 - D append DB to positive numbers.
 - E enclose negative numbers in angle brackets.
 - M append - sign to negative numbers.
 - N suppress the - sign.

Some examples, showing the source data, the mask and the results, are shown in Figure 16.21. I've used the ~ character to represent a space where it is significant.

Source	Mask	Result	Source	Mask	Result
12345	%3	123	-123.456	L2\$%10	\$-123.4600
ABCDE	%3	ABC	123.456	L2\$%10	\$123.46000

-123.456	2C	123.46CR	ABCDE	L2\$#10	\$ABCDE.000
123.456	2C	123.46~~	-123.456	L2%10	-123.46000
-123.456	2D	123.46~~	123.456	L2%10	123.460000
123.456	2D	123.46DB	ABCDE	L2%10	ABCDE.0000
-123.456	2E	<123.46>	-123.456	L2*10	-123.46***
123.456	2E	~123.46~	123.456	L2*10	123.46****
-123.456	2M	123.46~	ABCDE	L2*10	ABCDE.00**
123.456	2M	123.46~	12345	R#3	345
-123.456	2N	123.46	ABCDE	R#3	CDE
123.456	2N	123.46	123.456	R2#10	~~~~123.46
-123.456	2	-123.46	ABCDE	R2#10	~~ABCDE.00
-123.45	2	-123.45	-123.456	R2\$#10	00\$-123.46
-12345	2	-12345.00	123.456	R2\$#10	000\$123.46
123.456	2	123.46	ABCDE	R2\$#10	0\$ABCDE.00
123.45	2	123.45	-123.456	R2%10	000~123.46
12345	2	12345.00	123.456	R2%10	0000123.46
ABCDE	2	ABCDE.00	-123.456	R2*10	***123.46
12345	L#3	123	123.456	R2*10	****123.46
ABCDE	L#3	ABC	ABCDE	R2*10	**ABCDE.00
-123.456	L2#9	-123.46~~	12345	R3	12345.000
123.4445	L2#9	123.44~~~	ABCDE	R3	ABCDE.000
123.4449	L2#9	123.44~~~	ABCDE	R=10	=====
123.445	L2#9	123.45~~~	9876	D	14-Jan-1995
123.446	L2#9	123.45~~~	9876	D2/	14/01/95
123.456	L2#9	123.46~~~	9876	D2	14-Jan-95
ABCDE	L2#9	ABCDE.00~	9876	DWA	Saturday

Figure 16.21: Format masks

An important difference between the formatting of numeric data by means of a format mask, as here, and the conversion of data by means of the ICONV/OCONV function is that a mask retains the *actual decimal point of the number*, whereas the ICONV/OCONV function assumes that the number is an integer value and *imposes a decimal point on the number*.

Format masks may also be used in assignment statements to convert the appearance of the data as it is moved to the destination variable. For example, the sequence:

```
KEY = 123
KEY = KEY 'R%6'
```

would leave the string 000123 in KEY.

16.7.2 @ function

The @ function generates a string which, when used with the PRINT or the CRT statement, will move the cursor to a particular position on the screen or produce some other display effect. Here are some examples:

```
CRT @ (0,23): 'End of job'
```

move the cursor to column 0 (the left) of line 23 (the bottom) of the screen, hold the cursor there and then output the message: End of job

```
CRT @ (5,6) : 'NAME' : @ (20) : 'TELEPHONE'
```

move the cursor to column 5 of line 6, display the word NAME, move the cursor to column 20 (still on line 6 because the line specification is omitted), and display the word TELEPHONE.

The general forms of the function are:

```
@ (column, row)
```

and:

@(column)

The top of the screen is row 0, and the left of the screen is column 0.

16.7.3 @ function with negative argument

A further form of the @ function has a single negative argument:

@(-n)

and produces any of a number of special display effects. For example:

CRT @(-1)

clear the screen.

CRT @ (12,12):MESSAGE:@(-3):

display a message at the centre of the screen and then clear the rest of the screen (below and left of the message).

Some of the available arguments for the @ function are shown in Figure 16.22. The same forms are used with the Proc T statement, as discussed in Chapter 17. EPick gives full details of the forms of the @ function which are available on your system.

@(-1)	clear the screen and move the cursor to the top left corner of the screen.
@(-2)	move the cursor to the top left corner of the screen.
@(-3)	clear the rest of the screen (below and right of the cursor position).
@(-4)	clear the rest of the line (right of the cursor position).
@(-5)	make the rest of the screen blink, and
@(-6)	stop the rest of the screen blinking.
@(-13)	enable reverse video and
@(-14)	disable reverse video.

Figure 16.22: @ function with negative arguments

In general, the @ function is used to control *screen output*, and a statement such as:

```
PRINT @ (5,5):NAME:
```

will not produce the desired effect of positioning the text when sending output to the printer. There are, however, certain forms of the @ function which *can* be used with the printer, such as those which download printer fonts and impose other settings on the printer. The System Manager can extend these and the other @ functions by using the DEFINE-TERMINAL command to make the appropriate changes to the items on the DEVICES file which control each type of terminal and printer.

16.7.4 PRINTER ON

The PRINT statement normally displays the output on the screen. If you wish to **print** your results on the printer, then you must first activate the **spooler** by using the:

```
PRINTER ON
```

statement. All subsequent PRINT statements will then send their output to the printer (strictly, we should say **to the spooler**); this continues until the program terminates or until you execute the:

```
PRINTER OFF
```

statement. The actual printout is produced only when the spooler job is closed; this will be when the program terminates or when the:

```
PRINTER CLOSE
```

statement is executed by the program.

If you invoke a program with the P option:

```
RUN MBPROGS WAGES001 (P
```

or, if the program has been catalogued and you invoke it with a command such as:

```
WAGES001 (P
```

then the P option has the same effect as having a PRINTER ON statement at the start of the program and sends all PRINT output to the spooler. In all cases, any CRT statements in the program are not affected by this and **CRT statements always display their output on the screen**; this is the chief difference between the PRINT and CRT statements.

16.7.5 PRINT ON

A useful form of the PRINT statement allows you to produce *several*

distinct printed reports at the same time. For example, the statements:

```
PRINT ON 1 ORDER, ITEM, DESCRIPTION, QUANTITY
PRINT ON 2 DESCRIPTION, QUANTITY
```

will send one set of output to print-file #1 and the other to print-file #2. At the end of the program (or when a **PRINTER CLOSE** statement is executed), all the output sent to print-file #1 will come out together, then all the output sent to print-file #2, and so on. If you have issued **TCL** commands such as:

```
SP-ASSIGN R1 F3
SP-ASSIGN R2 F7
```

prior to executing the program, then all the output which the program sends to print-file #1 will go form-queue 3, and all that sent to print-file #2 will go to form-queue 7. Depending upon the way in which the System Manager has set up your spooler system (as we discuss in Chapter 19), the print-files may go to different printers.

Any **PRINT** statements without the **ON** element are considered to refer to print-file #0 and will be printed first.

16.7.6 HEADING / FOOTING statements

Headings and footings may be produced when your program is printing or displaying a report. These have the same features and form as the headings / footings used in Access and other areas of Advanced Pick. For example:

```
HEADING "'C'* * Daily Orders '*LL'"
```

declare a heading to be printed at the top of all subsequent pages.

```
FOOTING "'L'Press <Return> for next page"
```

declare a footing to be printed at the bottom of all subsequent pages.

Embedded within the text of the message, you may include any of a number of special *options* enclosed in apostrophes. The most important of these options include:

- · output a single apostrophe.
- C output the text in the centre of the line.
- D output today's date.
- L start a new line.
- PN output the page-number.
- T output today's date and the time.

Other options are shown in the Appendix.

When a **HEADING** and/or **FOOTING** is executed, the system will first clear the screen, output the heading and then automatically start to count the lines which are output. When a full page has been produced (typically 24 lines per screen), it will output the footing and wait for you to

hit <Return> before starting a new page and resetting the line counter. If the output is going to the printer (because a PRINTER ON statement has been executed), then the action will be identical, except that the page length will be different (typically 60 lines per pag) and it will not pause at the foot of each page.

If you wish to force a skip to a new page whilst the output is controlled by HEADING / FOOTING statements, you will use the:

```
PAGE
```

statement. This will output the footing and/or the heading and zeroise the line counter. The extended form:

```
PAGE 1
```

or:

```
PAGE CNTR
```

will reset the page number which appears in the heading / footing. The SYSTEM(4), SYSTEM(5) and SYSTEM(6) functions shown in Figure 16.18, can be used when a HEADING/FOOTING is active in the program.

It is important to note that the line count which controls the pagination is only incremented by PRINT statements which do not end in a colon. Thus, statements such as:

```
PRINT NAME:
CRT NAME
CRT NAME:
OUT NUMVALUE
```

will not increment the counter, whilst:

```
PRINT NAME
PRINT NAME, AGE
PRINT
PRINT 'The answer = ':RESPONSE
```

will increment the counter.

Although only one heading and one footing may be in action at one time, you may change your HEADING / FOOTING as often as required, and each will become effective at the next page break. Advanced Pick has no formal means of cancelling a heading / footing, and you must be particularly conscious of this if you are mixing printer output and screen output using the PRINT statement, since any heading / footing which you may have set for a printed report will still be effective when you go back to PRINT output on the screen. One means of disabling a heading / footing which you set earlier is to use a form such as:

```
HEADING "'N'"
FOOTING "'N'"
```

in which the N option suppresses the end-of-page pause on screen

output; without this, you would simply impose a null heading / footing and the pagination control would still apply.

16.8 Keyboard input

Most programs have some means of interacting with the user. This is often conversational in nature, displaying questions on the screen and accepting the user's response from the keyboard. The INPUT statement is used to accept data typed in at the keyboard as the program is executing and store the input in a variable. The program in Figure 16.23 uses the INPUT statement to ask the user for a series of numbers and, when he/she finally enters END, the program displays the total.

```
CRT @(15,0):'** Add up a series of numbers **'
TOTAL=0
LOOP
    CRT 'Enter the next number / END: ':
    INPUT NUMBER
    UNTIL NUMBER='END' DO
        TOTAL=TOTAL+NUMBER
    REPEAT
CRT
CRT 'The total is ':TOTAL
```

Figure 16.23: INPUT statement

When the program executes the statement:

```
INPUT NUMBER
```

the program will halt and wait for the user to type in a value, a number or a string, and press <Return>; the value will then be placed in NUMBER so that it can be used by the program. The data is placed in the variable exactly as entered by the user, including any spaces.

Normally, the program will display a question mark to indicate that it is waiting for input. You can use the PROMPT statement:

```
PROMPT '':'
```

to change this to any other character. You can even cancel the prompt character altogether:

```
PROMPT ''
```

in which case you would probably include all the text and punctuation in your prompt messages:

```
PRINT @(0,5):'Enter your name: ':
INPUT NAME
PRINT @(0,6):'How old are you? ':
INPUT AGE
```

You can only accept *one* piece of information (that is, a single numeric value or a single string) in each INPUT statement. If you want to ask the user for several pieces of data, then you must use several INPUT statements:

```
INPUT NAME
INPUT INITIALS
INPUT TELEPHONE.NUMBER
INPUT SCORE(ROW,COL)
INPUT INCOME(5)
```

The full form of the INPUT statement is:

```
INPUT variable {,n} {;} { } {FOR t {THEN/ELSE ..}}
```

where *n* indicates the maximum number of characters which may be entered (a <Return> will be assumed when *n* characters have been typed); the colon indicates that the cursor is to be held in position after the last character entered; the underscore is used with the length specification and indicates that the user must hit <Return> to send the data; *t* indicates that the processor is to wait for *t* tenths of a second for the input. The time limit parameter forces a **time-out** and the THEN clause is executed if the user enters the data within the time limit, the ELSE clause is taken otherwise. An example of this form might be:

```
Ask: INPUT NAME FOR 50 ELSE GOSUB Remind; GO Ask
```

which will wait for five seconds and then jump to a subroutine to remind the user that input is required (possibly sounding the terminal bell) before returning to wait for the input.

16.8.1 DATA statement

The DATA statement has forms such as:

```
DATA 'YES'
DATA 'SMITH', '15/6/53', 'MALE'
```

and will place a value, or sequence of values, in a queue from which they will taken, one at a time, by the next operation(s) which requires input from the keyboard. This is particularly useful when you are testing a program which uses the INPUT statement to accept data from the keyboard.

To illustrate this, let us imagine that you are writing a program which requires two pieces of data from the user: a file name, and an item-id. The fragment in Figure 16.24 shows how we might use the DATA statement to feed the INPUT. The DATA statement places two strings – STOCK and 1000 – in the data queue, and these are picked up in turn by the two INPUT statements. This avoids your having to type in the same data each time you test the program. When the program has been tested, the DATA statement can be removed and the INPUT statements will then come to the keyboard for their data.

```
DATA 'STOCK',1000
* Get file name
  CRT 'File name: ':
  INPUT FILE.NAME
  OPEN FILE.NAME
* Get ITEM ID
  CRT 'Item id: ':
  INPUT ITEM.ID
  READ RECORD FROM FV, ITEM.ID
```

Figure 16.24: DATA and INPUT

16.8.2 IN / OUT statements

Like some R83 Pick releases, Advanced Pick offers two further statements for handling input and output to the terminal: IN and OUT. These are valuable when the terminal supports non-standard equipment such as bar-code readers and point-of-sales equipment.

The IN statement has a form such as:

```
IN XCODE
```

and this will accept a *single character* from the keyboard. Thus, if the user responded by typing the letter:

```
A
```

the value 65 (that is, the decimal value of the ASCII A character) would be placed in XCODE. When you use the IN statement:

- o There is no prompt displayed,
- o The <Return> key is not required.
- o The cursor remains in position immediately after the newly-displayed character.
- o Control characters, such as the <Return> key, the cursor-control keys and backspace, are acceptable as input characters. The <Backspace> key does not, in this context, have the effect of moving the cursor back.

The OUT statement does the opposite. Thus, the statement:

```
OUT VALUE
```

will take the numeric value held in VALUE and convert this to the equivalent ASCII character and send this to the terminal. So, if VALUE contained the number 65, this statement would display the letter A on the screen.

16.8.3 INPUT @ statement and friends

The INPUT @ statement is one of a group of statements which offer very

powerful facilities for data input. This has the general form:

```
INPUT @ (C,r) : variable {,n}{mask}{:}
```

where *c* and *r* specify the screen position (as in the @ function) at which the data is to be accepted; *mask* is any format mask which is to be imposed on the data as it is redisplayed on the screen. To illustrate this, let's consider these statements:

```
TAX=0
INPUT @ (5,12):TAX "R2#12"
```

When the INPUT @ statement is executed, the run-time processor will:

- o Move the cursor to position (5,12) on the screen (that is, column 5 of row 12).
- o Display the current contents of the variable TAX (0 in this particular sequence) according to the mask "R2#12", that is right-justified in a field of width 12 and with two places of decimals.
- o Move the cursor back to position (5,12).
- o Wait for the user to enter a new value.
- o Redisplay the new value at position (5,12) according to the mask "R2#12".
- o Assign the new value to the variable TAX.
- o If the user hits <Return> instead of entering a value, this means that the previous contents of TAX (0 in this instance) are to be retained. Thus, the contents of TAX before the statement can be thought of as a *default* value.

There are several statements to support the INPUT @ facility.

INPUTERR controls the display and erasure of error-messages at the foot of the screen. For example:

```
INPUTERR "VALUE MUST BE LESS THAN 100"
```

When this statement is encountered it will clear the bottom line of the screen, if there is already an INPUTERR error-message there, and display the message. Note that – despite the keyword – there is no *error-condition* associated with this statement and it may be used to display messages quite independently of the INPUT @ statement.

We have seen that a *null value* will indicate that the current contents of TAX are to be retained, but what if we want to allow the user enter a value of null? The **INPUTNULL** statement allows the programmer to specify a character which is to be used to represent a truly null value. For example, the statement:

```
INPUTNULL "♦"
```

is not executed immediately, but indicates that, if the user enters * at any subsequent INPUT @ statements, then this is to be interpreted as a *null value*. Any single character may be specified, and this null-indicator will stay in operation until it is changed by a subsequent INPUTNULL statement.

The INPUTTRAP statement has two forms and allows a set of special single-character codes to be input by the user. If one of these special characters is entered, the processing will jump to a specific statement label where the program will take the appropriate action. Some examples of this statement are:

```
INPUTTRAP "?/.<>" GOTO 10,20,30,40,50
INPUTTRAP "?/.<>" GOSUB 10,20,30,40,50
INPUTTRAP "?/.<>" GOTO Help,Scrub,Stop,Back,Forward
```

This is a useful feature which a Systems Analyst might exploit to design systems with user input conventions such as:

- ? to display a help message.
- / to cancel the operation.
- . to abandon the job.
- < to return to the last input stage.
- > to skip to the next input stage.

and so on. There may be any number of characters and a corresponding set of destinations, and these INPUTTRAPs will stay in operation until changed by a later INPUTTRAP statement. The INPUTTRAP and INPUTNULL statements have no immediate effect, they prepare the way in which any future INPUT @ statements will behave.

16.8.4 Case sensitivity

Whenever you are accepting data from the user, you should pay particular attention to the question of **case sensitivity**. Unless you have issued any special statements in your program, all data is **case insensitive**. For example, data which is handled by a program may be entered and held as:

```
CHAIR, RED, OAK
chair, red, oak
Chair, Red, Oak
cHaIr, rEd, oAk
```

and all four forms will be equivalent for the purposes of statements such as:

```
IF DESCRIPTION = 'CHAIN, RED, OAK' THEN STOP
IF INDEX(DESCRIPTION,'RED',1) THEN CRT 'RED'
IF INDEX(DESCRIPTION,'red',1) THEN CRT 'RED'
IF DESCRIPTION[1,2] = 'CH' THEN GOSUB DO.CH
```

This might be troublesome in certain situations. The Basic statements CASING ON / CASING OFF can be used to specify whether all

subsequent data strings, including keyboard input data and file data, are to be case sensitive or not. If CASING is OFF, then the four values shown here are considered to be identical; with CASING ON, the four will be different strings.

16.9 Subroutines

All programming languages have some means of organising a piece of coding in such a way that it may be used from several places within a program. There may even be a way of making the same piece of coding accessible to several programs. Other languages use terms such as *procedure*, *subprogram* and *module* to describe such a piece of coding, Basic uses the name **subroutine**. The language offers two types of subroutine: **internal subroutines** and **external subroutines**. The power to use external subroutines means that structured programming can be exploited in Basic programs on Advanced Pick.

16.9.1 Internal subroutines

Internal subroutines are sequences of code held within the program in which they are used and are compiled as a part of that program. I used three internal subroutines – called **Get.dataname**, **Vet.dataname** and **Execute.sentence** – in the SENTENCE01 program in Figure 16.1. Control is passed to the internal subroutines by statements of the form:

```
GOSUB Vet.dataname
```

The program then jumps to that statement label and continues executing from there. When a:

```
RETURN
```

statement is encountered, the processing jumps back to the main part of the program (at the statement immediately following the GOSUB statement). You may have several such GOSUB statements in a program to use the same subroutine from several places within the program. This is one advantage of using subroutines ... you only have to include the coding once, although you may use it many times; this reduces the maintenance if the processing within the subroutine is to be changed at any time.

16.9.2 External subroutines

If you wish to use the same coding in several programs, you could use an **external subroutine**. As a simple example, let us suppose that I want a subroutine which will ...

- ... clear the screen and
- ... display the date in the top left corner and
- ... display the time in the top right corner of the screen.

This is quite a useful routine and could be used in many different programs. The subroutine in Figure 16.25 shows how we might do this.

Note that the first line of this item consists of the word **SUBROUTINE** followed by the name of the subroutine.

```

01      SUBROUTINE SCREEN.MASK
02 * General subroutine to ...
03 * Clear the screen ...
04      CRT @(-1):
05 * Display the time ...
06      CRT @(0,0):OCONV(TIME(),'MTS'):
07 * Display the date
08      CRT @(SYSTEM(2)-7,0):OCONV(DATE(),'D2/')
09      RETURN

```

Figure 16.25: An external subroutine

The subroutine is a complete and independent item and will be created, amended and compiled just like a normal program. When I need to use this subroutine in one of my programs, I use a **CALL** statement such as:

```

      CALL SCREEN.MASK
OF:
      CALL 'SCREEN.MASK'
OF:
      TARGET='SCREEN.MASK'
      CALL @TARGET

```

and the processing will jump to the **SCREEN.MASK** subroutine. When the **RETURN** statement in the **SCREEN.MASK** subroutine is executed, control will be passed back to the program which called it (returning to the statement immediately following the **CALL** statement). You can see this in action in the revised version of **SENTENCE01** which is shown in the first part of Figure 16.26.

But how does my program know where the coding for the **SCREEN.MASK** routine is to be found? It doesn't! So the run-time processor always looks to the MD and expects to find an item there called **SCREEN.MASK** which records the name of the file and the item where the **SCREEN.MASK** subroutine is to be found. I must create this **program pointer** item by means of a command such as:

```
CATALOG MBPROGS SCREEN.MASK
```

when I have compiled the subroutine. We'll say more about cataloguing later, but its principal action is to set up – on the MD – a pointer to the name of the file and the subroutine.

16.9.3 Passing data to external subroutines – 1

The **SCREEN.MASK** subroutine is a fairly simple subroutine since it contains all the data which it needs. It does not share any data with the

main program. But if I wanted to reorganise my SENTENCE01 program to implement the GET.DATANAME, VET.DATANAME and EXECUTE.SENTENCE routines as **external subroutines**, I should need some way of passing information – such as the DATANAME and the SENTENCE – between the main program and the subroutines. Look at Figure 16.26 and see how this is done. The statement:

```
CALL GET.DATANAME(COUNT,DATANAME)
```

passes control to the GET.DATANAME subroutine and carries with it the contents of COUNT and DATANAME. When the processing arrives at the GET.DATANAME subroutine, these two values tie up with those on the statement:

```
SUBROUTINE GET.DATANAME(COUNT,DATANAME)
```

and within the subroutine, I can use and change the contents of the variables COUNT and DATANAME; these are known as **local variables**. When the RETURN statement is executed, the contents of the local variables COUNT and DATANAME are transferred back to the main program, thus passing the DATANAME, which was input in the subroutine, back to the main program. I use another external subroutine to validate the dataname in VET.DATANAME, and another, EXECUTE.SENTENCE, to execute the final Access sentence.

```
01 * Program to construct simple Access sentences
02
03 OPEN 'DICT STOCK' TO STOCK.DICT ELSE STOP
04
05 PROMPT ''
06
07 EQUATE DN TO CHAR(10)
08 COUNT = 1
09 SENTENCE = 'SORT STOCK'
10 *****
11 CALL SCREEN.MASK
12 CRT @(25,0) : '*** Access sentence builder ***'
13
14 LOOP
15     CALL GET.DATANAME(COUNT,DATANAME)
16 UNTIL DATANAME = '.' DO
17     CALL VET.DATANAME(STOCK.DICT,DATANAME,SENTENCE,COUNT)
18 REPEAT
19
20 CALL EXECUTE.SENTENCE(SENTENCE)
```

```

-----
01      SUBROUTINE GET.DATANAME(COUNT,DATANAME)
02 * Ask for the dataname
03
04      EQUATE DN TO CHAR(10)
05      MSG1 = DN : 'Enter dataname #'
06      CRT MSG1 : COUNT : ' ':
07      INPUT DATANAME
08      DATANAME = TRIM(DATANAME)
09      RETURN
-----

01      SUBROUTINE VET.DATANAME(STOCK.DICT,DATANAME,SENTENCE,COUNT)
02 * Check that the dataname exists
03
04      EQUATE DN TO CHAR(10)
05      MSG2 = DN : 'Cannot find dataname '
06      MSG3 = 'Enter a STOCK dataname or . if no more'
07      READ DEFN FROM STOCK.DICT,DATANAME ELSE GO Missing
08      IF DEFN<1> NE 'A' AND DEFN<1> NE 'S' THEN GO Missing
09 Add.dataname:
10      SENTENCE = SENTENCE : ' ' : DATANAME
11      COUNT = COUNT + 1
12      RETURN
13 Missing:
14      CRT MSG2 : DATANAME
15      CRT MSG3
16      RETURN
-----

01      SUBROUTINE EXECUTE.SENTENCE(SENTENCE)
02 * Ask whether to execute the sentence
03      EQUATE DN TO CHAR(10)
04      CRT DN : 'Executing ' : SENTENCE
05      CRT 'Proceed? Y/N? ' :
06      INPUT RESP,1
07      IF RESP='Y' THEN EXECUTE SENTENCE
08      RETURN

```

Figure 16.26: External subroutines

In the case of the subroutines used here, it is unlikely that they would be

of use in any program other than SENTENCE01, but it still highlights an important advantage of constructing your programs as a set of external subroutines: it allows the programming effort to be split up into smaller, more manageable tasks, possibly allocating the subroutines to several programmers; the complicated processes can be written by more experienced programmers, the simpler tasks – such as SCREEN.MASK – might be left to junior programmers. Furthermore, If I ever need to modify, say, the SCREEN.MASK subroutine, I can do this quite easily and independently without having to change the SENTENCE01 program or the other subroutines.

When you are passing data as the **arguments** or the **parameters** of a subroutine in this way, you must be sure that the *number of parameters*, and the *order* in which you present them, are consistent on both the CALL statement and the SUBROUTINE statement. They don't have to have the same names in the main program and the subroutine, (although they frequently do). For example, I could have used the name XXXXX instead of SENTENCE throughout the EXECUTE.SENTENCE subroutine, and the statement:

```
CALL EXECUTE.SENTENCE(SENTENCE)
```

would have linked perfectly well with the statement:

```
SUBROUTINE EXECUTE.SENTENCE(XXXXX)
```

A main program and each subroutine has its own **local variables** which are held with the code for that program or subroutine. When a CALL statement is executed, the data is passed from one set of local variables to another, and the data is transferred in the opposite direction when the RETURN statement is executed.

16.9.4 Passing data to external subroutines – 2

As the number of names in the parameter list increases, the time taken to pass control to and from the subroutines also increases. We can overcome this by using a set of **COMMON variables** which are directly accessible from the main program and all the subroutines. Figure 16.27 shows how this might be done. Note the following points:

- I've only shown those parts of the program and the subroutines which have changed. The rest is as before.
- The COMMON statement in the main program and the subroutines declares a list of variables which are to be held as COMMON – or **globally accessible** – variables. You can think of these variables as being held *outside* the program and not as local variables which would be accessible only to the individual program or subroutine.
- The CALL statement and the SUBROUTINE statement do not now use a parameter list to pass the data.
- The variables in the COMMON set are identified by their *position* within the set and not by name.

- o Most programmers use the same COMMON statement in the main program and all its subroutines. But if we don't need all the variables in a subroutine – as in GET.DATANAME – then we can omit those from the end of the COMMON list.
- o The EXECUTE.SENTENCE subroutine only uses the last of the four COMMON variables, so I have used dummy names X1, X2 and X3 to make SENTENCE the fourth variable.

```

01 * Program to construct simple Access sentences
02     COMMON COUNT, DATANAME, STOCK.DICT, SENTENCE
03     OPEN 'DICT STOCK' TO STOCK.DICT ELSE STOP
04
05     PROMPT ''
06
07     EQUATE DN TO CHAR(10)
08     COUNT = 1
09     SENTENCE = 'SORT STOCK'
10 *****
11     CALL SCREEN.MASK
12     CRT @(25,0) : '** Access sentence builder **'
13
14     LOOP
15         CALL GET.DATANAME
16     UNTIL DATANAME = '.' DO
17         CALL VET.DATANAME
18     REPEAT
19
20     CALL EXECUTE.SENTENCE

```

```

-----
01     SUBROUTINE GET.DATANAME
02 * Ask for the dataname
03     COMMON COUNT, DATANAME
04     EQUATE DN TO CHAR(10)
05     MSG1 = DN : 'Enter dataname #'
:

```

```

-----
01      SUBROUTINE VET.DATANAME
02 * Check that the dataname exists
03      COMMON COUNT, DATANAME, STOCK.DICT, SENTENCE
04      EQUATE DN TO CHAR(10)
05      MSG2 = DN : 'Cannot find dataname '
:
-----

01      SUBROUTINE EXECUTE.SENTENCE
02 * Ask whether to execute the sentence
03      COMMON X1, X2, X3, SENTENCE
04      EQUATE DN TO CHAR(10)
05      CRT DN : 'Executing ' : SENTENCE
:

```

Figure 16.27: Using external subroutines

There is a one-to-one correspondence between the variables named in the various COMMON statements: the first name, COUNT in the main program, corresponds with the first name, COUNT, in the subroutine, the second in the main program with the second in the subroutine, and so on. The names given to the COMMON variable need not be the same in the program and the subroutines, provided that the *order* in which you present them is consistent on the various COMMON statements.

Figure 16.28 illustrates how the four variables COUNT, DATANAME, STOCK.DICT and SENTENCE held in the COMMON area can be visualised as being outside the program and the subroutines; these are **global variables**. When VET.DATANAME subroutine adds 1 to the variable which it knows as COUNT, this is the first COMMON variable and it is adding 1 to the variable which SENTENCE01 knows as COUNT. The variables which are used only in a single routine are *local variables*. There are no local variables used by SENTENCE01, the variable MSG1 is local to GET.DATANAME, the variables DEFN, MSG2 and MSG3 are local to VET.DATANAME, and the variable RESP is local to EXECUTE.SENTENCE.

This diagram shows how important it is to get the names of the variables on the COMMON statement in the *correct order*. Any variable not mentioned on a COMMON statement will be set up as a *local variable*. The *external* block of COMMON variables is set up when the main program starts executing, and this first COMMON statement determines the maximum size of the COMMON block.

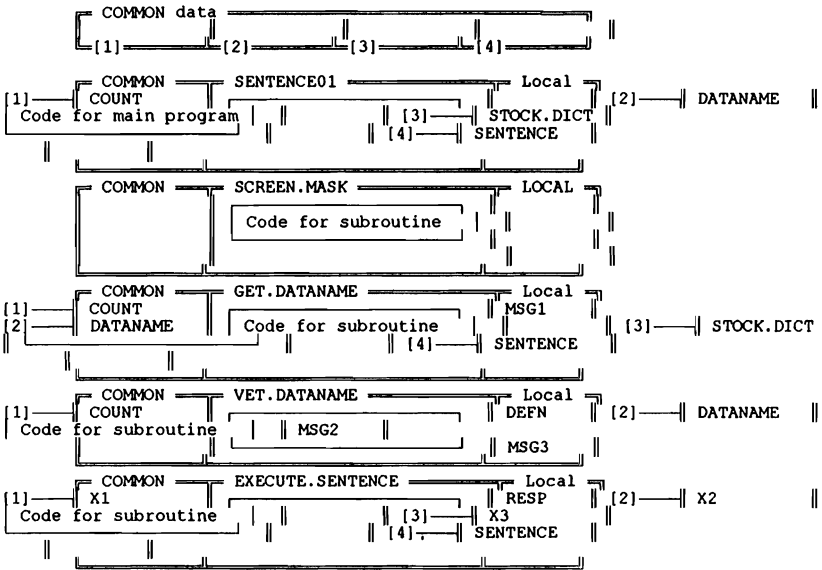


Figure 16.28: COMMON data

As with VET.DATANAME and EXECUTE.SENTENCE in this example, there are often occasions when a subroutine does not use all the COMMON variables. As with VET.DATANAME, it is easy to omit the last two names. But the subroutine EXECUTE.SENTENCE needs only SENTENCE; it would be wrong to use a statement such as:

```
COMMON SENTENCE
```

in EXECUTE.SENTENCE; if we did, then SENTENCE would actually point to the first COMMON variable (what the other routines know as COUNT). One way round this is to use dummy names:

```
COMMON X1, X2, X3, SENTENCE
```

as I did here, or even:

```
COMMON DUMMY(3), SENTENCE
```

16.9.5 Labelled COMMON statement

Advanced Pick offers a better solution known as **labelled common** or **named common**. This is illustrated in Figure 16.29 where I have used the special form:

```
COMMON /SCOMMON/ SENTENCE
```

to declare a labelled COMMON block called SCOMMON and which contains just the SENTENCE. Since EXECUTE.SENTENCE only uses this variable, I need only include the SCOMMON block there. I also used

SCOMMON in VET.DATANAME.

```

01 * Program to construct simple Access sentences
02     COMMON COUNT, DATANAME, STOCK.DICT
03     COMMON /SCOMMON/ SENTENCE
04     OPEN 'DICT STOCK' TO STOCK.DICT ELSE STOP
05
-----
01     SUBROUTINE GET.DATANAME
02 * Ask for the dataname
03     COMMON COUNT, DATANAME
04     EQUATE DN TO CHAR(10)
05
-----
01     SUBROUTINE VET.DATANAME
02 * Check that the dataname exists
03     COMMON COUNT, DATANAME, STOCK.DICT
04     COMMON /SCOMMON/ SENTENCE
05     EQUATE DN TO CHAR(10)
06
-----
01     SUBROUTINE EXECUTE.SENTENCE
02 * Ask whether to execute the sentence
03     COMMON /SCOMMON/ SENTENCE
04     EQUATE DN TO CHAR(10)
05

```

Figure 16.29: Using labelled COMMON

You may declare any number of variables and dimensioned arrays on a COMMON statement:

```
COMMON NAME, ADDRESS(5), TELEPHONE, AGE
```

and, having done this, you would not need a DIMENSION statement to identify ADDRESS as an array.

16.10 INCLUDE statement

If you wish to use the same statement(s) or the same piece of source code in several programs or subroutines, you can do this most easily by setting up the statements as a separate item and **including** this item in the programs and subroutines which need them. For example, I might create an item containing just the statement:

```
COMMON COUNT, DATANAME, STOCK.DICT, SENTENCE
```

and save this with an item-id such as SENT.COM on my MBPROGS file; then, instead of actually writing the COMMON statement in the program and the subroutines where I need to use it, I would simply use the statement:

```
INCLUDE SENT.COM
```

or, if the program or subroutine which uses the item is held on a file other than MBPROGS, I would use:

```
INCLUDE MBPROGS SENT.COM
```

When I compile a program which uses this INCLUDE statement, the compiler will read in the contents of SENT.COM and compile them just as if I had written the COMMON statement explicitly. Of course, if I change the contents of SENT.COM, I must recompile all the programs and subroutines which use it. The SENT.COM item is *not* compiled on its own.

For compatibility with other systems, you may use \$INCLUDE as an alternative to INCLUDE.

16.11 EXECUTE statement

The EXECUTE statement will let you issue any TCL command or Access sentence from within a Basic program. For example, the statement:

```
EXECUTE 'SORT STOCK WITH TYPE "DESK" LPTR'
```

will invoke an Access sentence and, when this is completed, the processing will return to the Basic program for any further work. The command to be executed may be specified as a literal or as an expression. Here are some more:

```
EXECUTE 'SSELECT STOCK WITH TYPE "DESK"'
EXECUTE 'SET-FLOPPY (AH)'
EXECUTE 'OFF'
EXECUTE 'SORT ':FILE:' WITH ':FIELD:' = '':VALUE:'' LPTR'
EXECUTE \SORT STOCK HEADING "'C'\:TITLE:\'L'" (P\
COMMAND='SORT ':FILE:' WITH ':FIELD:' '':VALUE:'' LPTR'
EXECUTE COMMAND
EXECUTE 'RUN BP AUDIT-REPORT' : CHAR(254) : 'OFF'
EXECUTE 'SSELECT STOCK' : CHAR(254) : 'SAVE-LIST XX'
```

The last two examples illustrate the way in which a sequence of

commands, separated by the attribute-mark, may be submitted for execution one after the other. This form can also be used to submit *data* to a process:

```
EXECUTE 'COPY STOCK *' : CHAR(254) : '(OLD.STOCK'
EXECUTE 'RUN BP CALC':CHAR(254):10:CHAR(254):100
```

This last example would execute the CALC program and then submit two data values 10 and 100 which might be picked up by INPUT statements in CALC.

As we saw earlier, the DATA statement offers an alternative means of submitting data to the EXECUTEd command:

```
DATA '(OLD.STOCK'
EXECUTE 'COPY STOCK *'
```

and the DATA statement can also be used to pass data to a program which is invoked by the EXECUTE statement:

```
DATA 10, 100
EXECUTE 'RUN BP CALC'
```

and these values will be passed to the first two INPUT statements which are executed within the CALC program.

16.11.1 CAPTURING clause

There are two extensions to the EXECUTE statement:

```
EXECUTE command {CAPTURING display} {RETURNING messages}
```

For example, in the statement:

```
EXECUTE 'WHAT' CAPTURING OPUT
```

used in Figure 16.30 (b), the CAPTURING clause specifies that the report which would be displayed by the command is not to be output, instead, it is to be **captured** and placed in a variable called OPUT; the contents of OPUT will be a dynamic array, one attribute for each line which would have been displayed on the screen.

16:29:59 29 Jul 1995							
memory	pibs	pcb0	sysbase/mod	maxfid	available	dfsize	
1664k	74	2048	2394	7	28536	2690	1000

(a) Part of the WHAT output at TCL

```
* Program to calculate percentage free space
```

```
EXECUTE 'WHAT' CAPTURING OPUT
```

```
DETAILS = TRIM(OPUT<3>); * 3rd line of display
```

```

TOTAL.FRAMES = FIELD(DETAILS,' ',6)
FREE.FRAMES = FIELD(DETAILS,' ',7)
FRAME.SIZE = FIELD(DETAILS,' ',8)
CRT 'There are ':FREE.FRAMES:' ':FRAME.SIZE:
CRT '-byte frames remaining out of ':TOTAL.FRAMES
FULL = (TOTAL.FRAMES-FREE.FRAMES)*100/TOTAL.FRAMES
CRT 'The system is ':FULL '2':'% full'

```

(b) The program

```

There are 2690 1000-byte frames remaining out of 28536
The system is 90.57% full

```

(c) The output

Figure 16.30: CAPTURING clause

You might use the CAPTURING clause when:

- You want to suppress the output which would otherwise be displayed on the screen. It may be that you don't want the user to see the display. We do this in Figure 16.31, where the user need not know the details of the file creation activity.
- You want to process the data which the command produces, as in Figure 16.30 where we capture the output from the WHAT report (the normal display is shown in (a)) and extract and interpret the information to tell the user how full the system is.

16.11.2 RETURNING clause

After a command has been EXECUTED, the RETURNING clause will return the **message numbers** (that is, the item-ids of the items on the MESSAGES file) of any messages which were produced by the command. You might use this to check whether or not the command executed successfully, as illustrated by the fragment in Figure 16.31

```

CRT 'Enter name for new file: ':
INPUT FILE.NAME
COMMAND = 'CREATE-FILE ':FILE.NAME:' 1 1'
EXECUTE COMMAND CAPTURING CCC RETURNING MSGS
IF INDEX(MSGS,'417',1) THEN
    GOSUB Successful
END ELSE
    GOSUB Failed
END

```

Figure 16.31: RETURNING clause

Here, we are attempting to create a file and need to know whether it worked or not. In this particular instance, the variables MSGS will contain:

```
417 417
```

if the file was created successfully (since the CREATE-FILE outputs

message 417:

```
[417] File 'xx' created; base = ff, modulo = mm
```

twice, once for each of the DICT and the data sections of the newly-created file).

16.11.3 TCL statement

The simple form of the TCL statement:

```
TCL 'SORT STOCK WITH TYPE "DESK" '
TCL 'SELECT STOCK' : CHAR(254) : 'SAVE-LIST XXX'
```

behaves exactly like the equivalent EXECUTE statement, and the extended form:

```
TCL command TO dimarr
```

behaves much like the statement:

```
EXECUTE command RETURNING string
```

except that the EXECUTE RETURNING clause returns a *string* of message numbers, whereas the TCL TO clause loads the elements of a *dimensioned array* with the number of the *last* message generated by the command and all the parameters which were included in that message. For example, the statement:

```
TCL 'CREATE-FILE STOCK2 3 37' TO DIMARR
```

would execute the command, possibly displaying a message such as:

```
[417] File 'STOCK2' created; base=9876, modulo=37
```

and loading the first four elements of the dimensioned DIMARR with the values 417, STOCK2, 9876, and 37. Compare this with the EXECUTE statement mentioned above.

16.11.4 CHAIN statement

If you do not want to return to your program after the command has been executed, you should use the CHAIN statement. This has forms such as:

```
CHAIN 'SORT STOCK WITH TYPE "DESK" LPTR'
CHAIN 'SORT ':FILE:' WITH ':FIELD:' ':VALUE:' " LPTR'
CHAIN COMMAND
CHAIN COMM1 : CHAR(254) : COMM2 : CHAR(253) : COMM3
```

and is slightly faster than the equivalent EXECUTE and TCL statements.

16.12 TCLREAD statement / ERROR() function

We have seen that the INPUT statement can be used to ask the user for

information as the program is executing. The TCLREAD statement offers an alternative way of accepting the user's data, allowing the program to retrieve the command which invoked the program, possibly picking up data from the TCL command line. For example, if we wanted a system whereby the user could type the TCL command:

```
RUN MBPROGS WAGES M
```

to produce the monthly wages, or:

```
RUN MBPROGS WAGES W
```

to produce the weekly wages, then the statement:

```
TCLREAD COMMAND
LETTER = FIELD(TRIM(COMMAND), ' ',4)
```

within the WAGES program could be used to place the entire TCL command as a string in COMMAND, and then extract the code M or W in the normal manner: .

If any options are specified on the TCL command:

```
RUN MBPROGS WAGES M (P
```

these will have their normal effect (printing any PRINT output in this instance) and they will also appear in the string returned by the TCLREAD statement exactly as they were entered. No matter whether or not the TCLREAD statement has been used, you can also obtain the options directly with the SYSTEM(15) function call. This returns all the options on the command sorted into alphabetical order and removes any duplicates and any non-alphabetic characters.

The oddly-named ERROR() function offers an alternative means of obtaining the TCL command which invoked the program. The two statements:

```
TCLREAD COMMAND
COMMAND = ERROR()
```

are identical in action.

If a program has been invoked by an EXECUTE statement in another program, the TCLREAD statement and/or the ERROR() function will return the command which was EXECUTED.

The effect is the same – and possibly more convenient – if the program has been catalogued and can be invoked by a TCL command such as:

```
WAGES M
WAGES W (P
```

Again, the entire TCL command may be obtained by the TCLREAD statement and/or the ERROR() function, and just the options by the

SYSTEM(15) function.

16.12.1 PROCREAD / PROCWRITE statements

The TCLREAD statement and the ERROR() function are new to Advanced Pick. Programmers who have moved from R83 Pick will be familiar with the older PROCREAD statement. This is available on Advanced Pick and is yet another means of obtaining the TCL command which invoked the program:

```
PROCREAD BUFFER THEN/ELSE ...
```

However, this assumes that the program has been invoked *from a Proc*, and places the contents of the Proc input buffer into the variable, called BUFFER in this instance. The ELSE clause is executed if the program was not invoked from a Proc. The use of PROCREAD is illustrated in Figure 16.32.

```
XXXXX
01 PQ
02 HRUN MBPROGS WAGES
03 P
```

(a) The Proc on the MD

```
:XXXXX M
```

(b) Calling the Proc at TCL

```
* Get the TCL command
  PROCREAD BUFFER ELSE STOP 'Must be called from a Proc'
  BUFFER = TRIM(BUFFER)
  LETTER = FIELD(BUFFER, ' ', 4)
```

(c) Part of the program WAGES on MBPROGS

Figure 16.32: Using PROCREAD

The PROCWRITE statement:

```
PROCWRITE NEWBUFFER
```

is available to load a string into the Proc input buffer; this would be used to pass information back to the Proc which invoked the program.

16.13 File handling

Now we come to a most important aspect of any programming language: the ability to handle data files. In Pick/Basic, this is relatively easy. The Basic language has been extended by adding:

```
OPEN
READ
WRITE
```

```
DELETE
CLEARFILE
```

statements which allow you to handle the data held on Pick files. You may handle the entire item as a *dimensioned array* or as a *dynamic array*, or you may handle a *single attribute* of an item.

When you are using an Advanced Pick tool such as the Update Processor or Access, you will refer to each file by a name such as STAFF or STOCK:

```
LIST STAFF WITH NAME "JONES"
LIST STOCK DESCRIPTION QUANTITY
```

and we can think of these names as the *external names* of the files. When you are using Basic to process the items on your files, you must first identify the files that you wish to use. This is achieved by means of the OPEN statement. The general form of this statement is:

```
OPEN file.name {TO file.variable} {THEN/ELSE ...}
```

For example, the statement:

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
```

will locate the file STOCK and store the location (the base FID and the modulo) of the file in a variable STOCK.FV in this instance, linking the external name of the file (STOCK) with the internal name (STOCK.FV). STOCK.FV is known as the **file-variable**. Some other examples are:

```
OPEN 'DICT STOCK' TO DICTVAR THEN GOTO 90
OPEN 'DICT', 'STOCK' TO DICTVAR THEN GOTO 90

FILE='STAFF'
OPEN FILE TO STAFF THEN GOSUB 100 ELSE GOSUB 200
```

A program may handle as many files as you need, and each one will be opened to its own file-variable. If your program is using only a single file, it is possible to omit the file-variable, using statements such as:

```
OPEN 'STOCK' ELSE STOP
READ RECORD FROM ITEMID ELSE GO 100
WRITE RECORD ON WORKID
DELETE ITEMID
CLEARFILE
```

but this can become confusing – even dangerous – if you are using more than one file.

The OPEN statement has a THEN/ELSE structure, and this has the same format – single line or multiline – that we met when we looked at the IF statement. The THEN clause is executed if the file can be found and the ELSE clause if the file cannot be found. You may omit the THEN/ELSE clause:


```
OPEN 'STOCK' TO STOCK.FV
OPEN 'STOCK'
```

in which case the compiler will interpret these as if you had written:

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP,201,'STOCK'
OPEN 'STOCK' ELSE STOP,201,'STOCK'
```

terminating the program with error-message 201:

```
'STOCK' is not a file name
```

if the file cannot be opened.

Having opened the file, you may now read items from the file, write items to the file and/or delete items from the file.

16.13.1 Reading data from a file

When you want to read data from a file, you will use the READ statement to read each *item* in turn, or you may read an individual *attribute* of an item. There are several forms of the READ statement. The simplest is READ, and this has the general form:

```
READ variable FROM file.variable, item.id THEN/ELSE ...
```

For example:

```
READ STOCK.ITEM FROM STOCK.FV, 1234 ELSE GOTO 10
READ STOCK.ITEM FROM STOCK.FV, '1234' ELSE STOP

ID=1234
READ STOCK.ITEM FROM STOCK.FV, ID ELSE GOTO 10

STOCK.ID=1234
READ STOCK.ITEM FROM STOCK.FV, STOCK.ID THEN
  GOSUB Found
END ELSE
  GOSUB Missing
END
```

The action of all these particular READ statements is to go to the file which has been opened to the file-variable STOCK.FV and read the item with the item-id 1234. If the item exists, then the entire contents of the item (attribute-marks and all, but not the item-id) will be placed into STOCK.ITEM and the THEN clause (if any) will be executed; if the item cannot be found, the program will execute the ELSE clause. The READ statement places the data in STOCK.ITEM as a *dynamic array*. Figure 16.33 illustrates how the READ statement transfers data from the physical item on disk to the STOCK.ITEM variable. In these and all our examples, the names which you use, STOCK.FV, STOCK.ID, STOCK.ITEM and so on, are chosen quite arbitrarily, but it's good practice to use names which remind you of the significance of the variables.

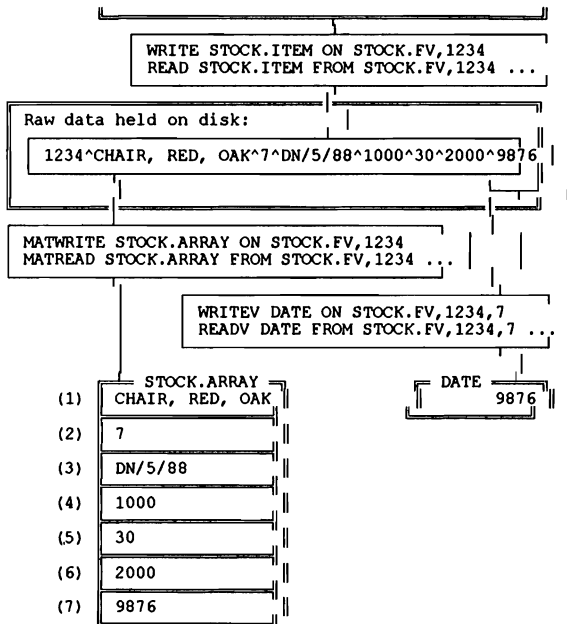


Figure 16.33: The READ / WRITE statements

As with the OPEN statement, you may omit the THEN/ELSE clause on the READ statement:

```
READ RECORD FROM STOCK.FV, ITEMID
```

in which case the compiler will interpret this as if you had written:

```
READ RECORD FROM STOCK.FV, ITEMID ELSE RECORD=''
```

returning an empty string if the item cannot be found. This applies to all forms of the READ statements described below.

The MATREAD statement:

```
MATREAD STOCK.ARRAY FROM STOCK.FV, 1234 ELSE GO 10
```

has a similar action to READ except that the contents of the item (without the item-id) will be placed in the *dimensioned array* called STOCK.ARRAY: attribute 1 (value-marks, subvalue-marks and all) will go into STOCK.ARRAY(1), attribute 2 will go into STOCK.ARRAY(2) and so on, as illustrated in Figure 16.33. If the array is larger than the item (that is, there are more elements in the array than attributes in the item), then the final elements will be null; if the array is smaller than the item, then the final attributes will be placed in the last element of the array as a *dynamic array*.

The third READ statement, the READV statement, has the form:

```
READV DATE FROM STOCK.FV, 1234, 7 ELSE GOTO 10
```

and will go to the file opened to the file-variable STOCK.FV, but it will only retrieve a *single attribute* – attribute 7 in this example – from the item with the item-id 1234 and place the data into DATE. This is illustrated in Figure 16.33. If the attribute contains values and/or subvalues, these will be placed in DATE as a dynamic array.

16.13.2 Processing the file data

When you have read data from a file into your program, you may process it as you wish: you can replace attributes with new values (a stock description may change); you can perform arithmetic on numeric attributes (goods may have been sold and the quantity in stock reduced, or goods may have been received and the quantity in stock increased). There are many possibilities.

If you are handling the file by means of READ/WRITE statements, then you will process the data as you would process any dynamic array:

```
PRINT 'THE PRICE IS ':STOCK.ITEM<4>
STOCK.ITEM<1> = 'THIS IS THE NEW TITLE'
STOCK.ITEM<2> = STOCK.ITEM<2>+DELIVERIES-ISSUED
PRINT STOCK.ITEM<7,3>
PRINT STOCK.ITEM<7,2,3>
```

If you are handling the file by means of MATREAD/MATWRITE statements, then you will process the data as you would process any dimensioned array:

```
PRINT 'THE PRICE IS ':STOCK.ARRAY(4)
STOCK.ARRAY(1) = NEW.DESCRPTION
STOCK.ARRAY(2) = STOCK.ARRAY(2)+DELIVERIES-ISSUED
```

If you need to handle the *values* and *subvalues* within a dimensioned array, then you will use references of the form:

```
PRINT STOCK.ARRAY(7)<1,3>
```

to handle the third multivalue of attribute 7, and:

```
PRINT STOCK.ARRAY(5)<1,2,3>
```

to handle the second *subvalue* of the third value of attribute 5. In all such cases, the attribute number will always be 1.

If you are handling the file by means of READV/WRITEV statements, then you will process the data as you would process a single piece of data, and, if you want to handle *multivalues* within such a variable, you will use references of the form:

```
PRINT DATE<1,3>
PRINT DATE<1,2,3>
```

16.13.3 Writing data to a file

When you have created a new item or amended an existing item, you may want to write it to the file. You will do this by means of the WRITE statement. Just as there are several READ statements, so there are several corresponding forms of the WRITE statement. Exactly which WRITE statement you use depends upon how the data is held by the program:

```
WRITE RECORD ON STOCK.FV, STOCK.ID
WRITE '*REMOVED*' ON STOCK.FV, 'AB/1000'
```

The WRITE statement will regard the source data as a dynamic array and use this to create the item on the file.

```
MATWRITE STOCK.ARRAY ON STOCK.FV, STOCK.ID
MATWRITE STOCK.ARRAY ON STOCK.FV, 'AB/1000'
```

The MATWRITE statement will use the contents of the dimensioned array to create the item, each element of the dimensioned array producing one attribute of the item. Any trailing null elements in the dimensioned array will be ignored.

```
WRITEV DATE ON STOCK.FV, STOCK.ID, 7
WRITEV LAST.INV.NO ON DICT.FV, '*LAST.INVOICE*', 3
WRITEV NEW1 ON PROG.FV, NAME, 0
WRITEV APPENDED ON PROG.FV, NAME, -1
```

The WRITEV statement will replace the specified attribute, inserting attribute-marks as necessary, leaving all other attributes in the item unchanged. Note the two special forms of the WRITEV statement:

```
WRITEV NEW1 ON FILE.FV, ITEMID, 0
    insert a new attribute at the front of the item and shift all the attributes up.
```

```
WRITEV LAST1 on FILE.FV, ITEMID, -1
    append a new attribute to the end of the item.
```

The general action of all the WRITE statements is the same: if the item already exists, the new version of the item will *overwrite* the previous version held on disk; if the item does not exist, a new item will be *created* on the file. Note that, unlike some other systems, Pick does not distinguish between *WRITE* and *rewrite*.

The decision as to which READ (and which WRITE) statement you should use and how you should process the data depends upon several factors: how much data you want from the file (a single attribute or all the item); how much processing you want to perform on that data (MATREAD is better if there is a lot of processing on several fields); whether you can predict the approximate number of attributes in the items (READ is better if you are building items which have an unknown number of attributes). It is quite acceptable – if unusual – to use one form of the READ statement to read data from the file, and a different form of the WRITE

statement to write data back to the file. If in doubt, you could do worse than use MATREAD / MATWRITE.

16.13.4 Deleting items from a file

Items are removed from the file by means of the DELETE statement. This has a form such as:

```
DELETE STOCK.FV, 1234
```

OR:

```
DELETE STOCK.FV, STOCK.ID
```

If there is no item with the item-id 1234 on the file which has been opened to file-variable STOCK.FV, then no action will be taken. If the item does exist, then it is deleted from the file. There is no *undelete* facility, so take care.

To delete all the items from a file, you will use the statement:

```
CLEARFILE STOCK.FV
```

Take special care with this one.

16.13.5 Batch processing

In a great many situations, a program will ask the user for the item-id of the item which is to be handled and then read and process that item:

```
CRT 'Enter item-id: ':  
INPUT STOCK.ID  
MATREAD STOCK.ARRAY FROM STOCK.FV, STOCK.ID THEN ...
```

The use of a B-tree index to find item-ids via a secondary key is discussed in Chapter 15 when we look at file indexing on Advanced Pick.

All forms of the READ and WRITE statements require a specific and explicit item-id. There is no simple way in which Basic will allow you to say "*read me the next item on the file*" or "*create me a new item-id*". There will be times when you want to perform **batch processing**, handling all the items on the file or a range of items on the file. If the items have calculable item-ids such as 1000, 1001, 1003, 1020 and so on, then it may be a simple matter to write a loop such as:

```
FOR STOCK.ID = 1000 TO 2000
```

to derive the item-ids. But an alternative solution is available where the item-ids cannot be derived by any other means: the SELECT statement:

```
SELECT STOCK.FV TO STOCK.SV
```

produces a list of the item-ids of the items on the file and associates these with a **select-variable**, called STOCK.SV in this instance. The item-ids can then be retrieved from the select-variable one by one each time the READNEXT statement is executed:

```
READNEXT STOCK.ID FROM STOCK.SV ELSE STOP
```

The READNEXT statement obtains the next item-id from the list held in STOCK.SV and puts this into STOCK.ID and once it is in there we can use it to READ the item from the file, or in any way we wish. This is illustrated in Figure 16.34. The ELSE clause on the READNEXT statement is executed when the item-ids in the select-variable have been exhausted.

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
SELECT STOCK.FV TO STOCK.SV
DONE=0
LOOP
    READNEXT STOCK.ID FROM STOCK.SV ELSE DONE=1
UNTIL DONE DO
    READ RECORD FROM STOCK.FV,STOCK.ID THEN
        PRINT STOCK.ID '#10':
        PRINT RECORD<1> '#50':
        PRINT RECORD<4> 'R2#10'
    END
REPEAT
```

Figure 16.34: SELECT / READNEXT / READ

There may be any number of select-variables active at any one time. If your program is using only a single select list, it is possible to omit the select-variable:

```
SELECT STOCK.FV
READNEXT STOCK.ID ELSE DONE=1
```

In all cases, the SELECT statement presents *all* the item-ids on the file in the *physical* order in which they are held on the file. Using this technique alone, you cannot sort the item-ids into any specific sequence, nor can you impose any selection criteria to select, say, only those items which are made of PINE. There is no SSELECT statement in Basic.

There are two ways round this problem. The first is to issue the Access SELECT or SSELECT command at TCL *before* you execute the program:

```
:SSELECT STOCK WITH MATERIAL "PINE" BY STOCK.LEVEL
:RUN MBPROGS STOCK.REPORT
```

The second way is to invoke the Access SELECT or SSELECT command by means of the Basic EXECUTE statement within your program, as illustrated by the fragment in Figure 16.35. The Access sentence may include any sort specifications and any selection criteria.

```
OPEN 'STOCK' TO STOCK.FV ELSE STOP
EXECUTE 'SSELECT STOCK WITH MATERIAL "PINE" BY PRICE'
```

```

SELECT STOCK.FV TO STOCK.SV
DONE=0
LOOP
    READNEXT KEY FROM STOCK.SV ELSE DONE=1
UNTIL DONE DO

```

Figure 16.35: EXECUTE / SELECT / READNEXT / READ

You may use the **SECONDARY** keyword on the **READNEXT** statement if you wish to handle both the primary and secondary select-lists at the same time:

```

EXECUTE 'SELECT STOCK WITH MATERIAL "PINE"
EXECUTE 'SELECT STOCK WITH PRICE > "50" SECONDARY'
READNEXT KEY.PINE ELSE STOP
READNEXT KEY.50 SECONDARY ELSE STOP

```

16.13.6 Locking

Pick has the facility to **lock** an item when it has been read from a file. This is a security feature preventing several users from attempting to update the same item simultaneously. Locking is achieved by means of modified forms of the **READ** / **WRITE** statements:

```

READU STOCK.ITEM FROM STOCK.FV, STOCK.ID ...
MATREADU STOCK.ARRAY FROM STOCK.FV, 1234 ...
READVU DATE FROM STOCK.FV, 1234, 7 ...

```

Note the **U** (for *Update*) at the end of the keyword. This is available for all forms of the **READ** statement and the lock will remain until:

- the program writes the item back to the file with a **WRITE** / **MATWRITE** / **WRITEV** statement, or ...
- the program ends, or ...
- the lock is released by a statement such as:

```
RELEASE STOCK.FV, STOCK.ID
```

If you wish to rewrite an item but still keep it locked, then you will use the forms:

```

WRITEU STOCK.ITEM ON STOCK.FV, STOCK.ID
MATWRITEU STOCK.ARRAY ON STOCK.FV, 1234
WRITEVU DATE ON STOCK.FV, 1234, 7

```

If user A attempts to read an item which user B has locked, then user A's program will beep and wait until user B releases the locked item. This can be inconvenient, so a **LOCKED** clause has been added to the **READ** statements, as illustrated by the fragment in Figure 16.36.

```

READU STOCK.ITEM FROM STOCK.FV, STOCK.ID LOCKED
GOSUB Handle.lock
END THEN
GOSUB Found

```

```
END ELSE  
    GOSUB Missing  
END
```

Figure 16.36: The LOCKED clause

This enables the programmer to offer some course of action – in the **Handle.lock** subroutine here – if an item is found to be locked, possibly allowing user A to abandon that item or to wait a little longer.

16.13.7 FILE statement

The FILE statement is a new Advanced Pick facility which allows a Basic program to handle a file by means of the *datanames* on the dictionary.

Let us imagine that our STAFF file has the attribute definitions shown in Figure 16.36 (a) and the data items shown in (b). The program and output in Figure 16.36(c) illustrate some of the features of the FILE statement.

```

DICT STAFF CODE A/AMC TAG..... CONVERSIONS.. CORRELATIVES.. TP MAX

```

```

NAME          A          1 NAME                                T 15
START.DATE A          2 START DATE D                          R 11
ENTITLEMENT A          3 ENTITLEMENT MD1                      A3**10'/'5' R 5
                                [WEEKS]
HOLIDAYS      A          3 HOLIDAYS                            R 5

```

(a) Attribute definitions

```

STAFF..... NAME..... START DATE. HOLIDAYS ENTITLEMENT
                                [WEEKS]

```

```

1      SMITH      .      01 Mar 1987      15      3.0
2      JONES      .      25 Nov 1989      11      2.2
3      ANSELL     .      04 Jun 1984      27      5.4
4      ANDREWS    .      14 Apr 1971      30      6.0

```

(b) Data items

```

* Demonstration of the FILE statement

```

```

FILE STAFF

```

```

LOOP

```

```

    CRT 'Employee number':; INPUT STAFF.ID

```

```

UNTIL STAFF.ID = '.' DO

```

```

    READ STAFF FROM STAFF.ID

```

```

    CRT '      Employee name: ':STAFF(NAME)

```

```

    CRT '      Start date: ':STAFF(START.DATE)

```

```

    CRT '      Current entitlement: ':STAFF(HOLIDAYS):

```

```

    CRT ' days = ': OCONV(STAFF(ENTITLEMENT), 'MD1'):' weeks'

```

```

    STAFF(HOLIDAYS) = STAFF(HOLIDAYS) + 1

```

```

    CRT '      New entitlement: ':STAFF(HOLIDAYS):

```

```

    CRT ' days = ': OCONV(STAFF(ENTITLEMENT), 'MD1'):' weeks'

```

```

    CRT

```

```

    WRITE STAFF ON STAFF.ID

```

```

REPEAT

```

```

Employee number?1

```

```

      Employee name: SMITH

```

```

        Start date: 7000
Current entitlement: 15 days = 3.0 weeks
        New entitlement: 16 days = 3.2 weeks

Employee number?4

        Employee name: ANSELL
        Start date: 6000
Current entitlement: 27 days = 5.4 weeks
        New entitlement: 28 days = 5.6 weeks

Enter Employee: .

```

(c) Program and output

Figure 16.37: FILE statement

Note the following points:

- The FILE statement declares the file name explicitly and without quotation marks. Any other forms, including *pathnames*, would be rejected by the compiler.
- The form:

```
FILE STAFF, RATES, TAX.CODES
```

can be used to declare several files at the same time.

- The compiler incorporates a number of additional items in the compiled program to support the FILE statement:
 - A DIMENSION statement to declare a dimensioned array called STAFF.
 - An OPEN statement to open the data section of the file to a variable called FV.STAFF.
- The program uses references of the form:

```
STAFF (NAME)
STAFF (START.DATE)
STAFF (HOLIDAYS)
```

together with the attribute definitions NAME, START.DATE and HOLIDAYS on the DICT section of the STAFF file in order to determine the attribute number of those fields.

- References of the form:

```
STAFF (X)
```

cannot be used, since they would imply the existence of a dataname X on the DICT of the STAFF file. The compiler rejects these and any other unresolved datanames with the message:

Attribute name does not exist in the dictionary

- o Care should be taken when attempting to use numeric datanames. References of the form:

```
STAFF(1)
```

represent the first element of the dimensioned array, and not a dataname 1 on the DICT of the STAFF file. You must use the form:

```
STAFF('1')
```

if you wish to use dataname 1 from the DICT of the STAFF file.

- o When a dataname is used in the program, any correlative codes in the attribute definition are applied, but any output conversion codes are not. This can be seen with the START DATE which is held and displayed as an *internal date*. The correlative (A3*100/'5') which calculates the ENTITLEMENT in weeks is applied correctly when the holidays are increased; the conversion code (MD1) is not applied automatically, but I have used the OCONV function to show how it might be done.
- o The READ and WRITE statements use only the item-id and not the file-variable when reading data into the dimensioned array. The forms:

```
CLEARFILE FV.STAFF
READV NAME FROM FV.STAFF, ITEM.ID, 1
```

can make use of the file-variable.

When the FILE statement and STAFF(HOLIDAYS) references are used in a program, the details are **bound** into the program when it is compiled. If any of the datanames are changed, then the program must be recompiled.

16.14 Basic and backing storage

Your Basic programs can handle data which is held on the backing storage devices, that is, on magnetic tape and/or floppy diskette. As with all Advanced Pick backing storage files, the data will be read and written as a sequence of **fixed length records**. The length of the records is the **block-size** specified (or implied) on the T-ATT command.

Before attempting to use backing storage, the TCL SET- commands, T-ATT, T-DET and other commands must be used to identify and select the device externally to the program. These can be issued at TCL before you execute the program, or they may EXECUTED from within the program.

Basic offers a range of facilities for handling backing storage directly by means of statements such as:

```
READT variable THEN/ELSE ...
REWIND THEN/ELSE ...
WRITET variable THEN/ELSE ...
```

In all cases, the THEN clause is executed if the statement completes successfully, and the ELSE clause is executed if the device is not ready or attached and the normal warning messages are also displayed. If you wish to suppress the messages and perform your own error-handling, you may use the forms:

```
READT variable THEN/ONERR ...
```

The ONERR clause is executed if an error condition is detected, and the nature of the error can be determined by inspecting the value returned by the SYSTEM(0) function call.

The READT statement will read individual sequential records from backing storage:

```
READT record THEN/{ELSE/ONERR} ...
```

The ELSE clause on the READT statement is executed:

- o If the device is not attached, or,
- o If the device is not ready, or,
- o When an end-of-file marker is detected.

The data is returned as a string of ASCII characters. There is also the READTX statement which will read a record from backing storage and pass it to the program as a string of **hexadecimal** characters.

The READTL statement will read the tape label:

```
READT old.label THEN/{ELSE/ONERR} ...
```

The WRITET statement will write individual records to backing storage:

```
WRITET record THEN/{ELSE/ONERR} ...
```

The WEOF statement will write an end-of-file marker to backing storage:

```
WEOF THEN/{ELSE/ONERR} ...
```

The REWIND statement will rewind the backing storage diskette/tape to the loading mark:

```
REWIND THEN/{ELSE/ONERR} ...
```

The record used in the READT and WRITET statements, called RECORD in these examples, is simply a string of ASCII data (or hexadecimal data with the READTX statement). The length of the data is that of the block-size specified on the T-ATT command. If you are writing a string *shorter* than this block-size, it will be padded with trailing spaces; if you are writing a record *longer* than the block-size, it will be truncated and an error-message displayed.

16.15 TCL commands for Basic programs

There are several tasks to be performed in the development of Basic programs and external subroutines. Let's summarise these steps before we look at them more closely.

- 1) Create and maintain the **source program** using the Update Processor or the Editor:

```
UPDATE file.name program.name
EDIT file.name program.name
```

These tools are used to create and change a program such as SENTENCE01 held in the data section of the MBPROGS file, as illustrated in Figure 16.38 (c).

- 2) Compile the program to reveal any **syntax errors** or to produce the **object program**:

```
BASIC file.name program.name
COMPILE file.name program.name
```

If there are no syntax errors, these commands create the object program SENTENCE01 held in the DICT section of the MBPROGS file, as illustrated in Figure 16.38 (b). If there are any syntax errors, then the programmer will return to step (1) and correct these.

- 3) Test the program to reveal any **logical errors**:

```
RUN file.name program.name
```

This command executes the object program SENTENCE01 held in the DICT section of the MBPROGS file. If there are any logical errors, the programmer will return to step (1) and correct these.

The program can then be made available to the users. *External subroutines*, however, require an additional step:

- 4) Catalogue the subroutine to place a pointer to the subroutine on the MD where it can be reached by all programs which use that subroutine:

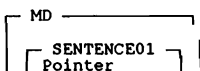
```
CATALOG file.name routine.name
```

This command sets up the pointer SENTENCE01 on the MD, as illustrated in Figure 16.38 (a).

If a *program* has been catalogued, it may be executed by the command:

```
program.name
```

as an alternative to the RUN command at step (3).



(a) Produced when the program

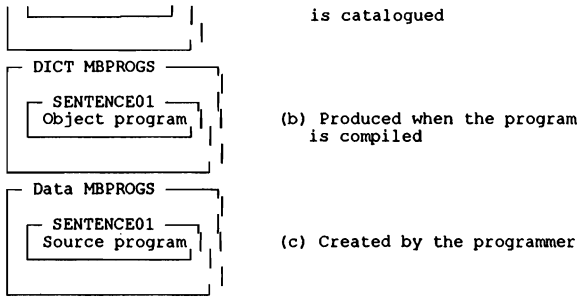


Figure 16.38: A Basic program

During the life of a program, the users will almost certainly require changes to a program. All these changes will be made by returning to step (1) and amending the *source program*.

Let's look more closely at these tasks.

16.15.1 BASIC / COMPILE commands

The purpose of both the BASIC verb and the COMPILE verb is to compile a **Basic source program** (or a Basic external subroutine), check for **syntax errors** and – if there are no errors – produce an equivalent **object program**. The two verbs are effectively identical; the only difference is that the BASIC verb is **case-sensitive** and distinguishes between UPPER-CASE and lower-case, whilst COMPILE is **case-insensitive** and does not distinguish between the two. Let's illustrate this:

`print total+TOTAL`
would be rejected by BASIC (because **PRINT** should be in upper-case) but it would be accepted by COMPILE and would refer to a single variable called **total**).

`PRINT total+TOTAL`
would be accepted by both BASIC and COMPILE, and would refer to two distinct variables (**total** and **TOTAL**) by BASIC, but a single variable (**total**) by COMPILE.

Beyond this case-sensitivity, the action and features of the BASIC and COMPILE commands are identical. The format of the commands is:

```

BASIC file.name item.list {(options)}
COMPILE file.name item.list {(options)}

```

In our examples here, we shall generally use the BASIC verb:

`BASIC SALES.PROGS ADD.RECORDS`
compile the program **ADD.RECORDS** which is held on the file **SALES.PROGS**.

`BASIC STOCK.BP ADD.STOCK AMEND.STOCK DELETE-STOCK`
compile three programs on the file **STOCK.BP** one after another.

BASIC STAFF.DB ADD.STAFF UPD.STAFF (PIL

compile two programs on the file STAFF.DB one after another, applying the P and I and L options.

BASIC RU.PROGS

BASIC RU.PROGS *

either of these would compile all the programs held on the file RU.PROGS.

SSELECT PROGS = "[AMEND]"

BASIC PROGS

select all programs which contain the word AMEND in their item-id and then compile these programs.

The same forms are available with the **COMPILE** verb.

Some of the options on the **BASIC** and **COMPILE** commands are shown in Figure 16.39.

- A** to output the Assembly language version of the program.
- C** to produce an object program without end-of-line markers. This produces a smaller object program but any compiler errors and/or **DEBUG** operations which refer to line-numbers will not be available.
- D** to prevent the use of the **D** option on the **RUN** command to enter the Basic debugger when the program is executed.
- E** to output the full text of any error lines as they are compiled.
- I** to output any sections of code which are brought in by means of an **INCLUDE** statement.
- L** to produce a listing of the source program as it is compiled.
- O** to produce optimised (Flash/Basic) object code.
- P** to send the compiler output to the printer.
- S** to suppress the production of the symbol table. This produces a smaller object program but any Basic debugger commands which

COMPILE and then RUN a Basic program. If several program names are specified, all programs are compiled, only the first is RUN.

COMPILE-CATALOG file.name item.list

COMPILE and then CATALOG a Basic program or subroutine.

If you are using the Update Processor to maintain your programs, you can use the <Ctrl> key sequences:

^XR file, and then COMPILE and RUN a Basic program, and

^XC file, and then COMPILE and CATALOG a Basic program or a subroutine.

Note that these commands and sequences use the COMPILE (case-insensitive) command to compile the program. There are no equivalent forms to call the BASIC compiler.

Later releases offer the LIST-OBJ utility to produce reports on the object programs on a file:

LIST-OBJ

display information about the object code on the BP file.

LIST-OBJ fffff

display information about the object code on file fffff.

16.15.3 RUN command

The purpose of the RUN command is to execute a *compiled* Basic program. The general form of the command is:

RUN file.name program.name {(options)}

For example:

```
RUN SALES.PROGS ADD.RECORDS
RUN STOCK.BP ADD.STOCK (D
RUN RU.PROGS DELETE-RECORD (P
```

Figure 16.41 shows some of the RUN options which are available.

A	abort the program execution if any fatal-error condition is detected, instead of entering the Basic debugger software.
D	enter the debugger at the start of program execution.
E	enter the debugger on any fatal/non-fatal error condition.
I	suppress the initialisation of variables at the start of the execution, leaving the storage area used by the variables and arrays in the

program as they were left by the last program which was executed on that port.

- N suppress the pause at the end-of-page when displaying HEADING / FOOTING controlled output on the screen.
- P direct the program output to the printer (spooler). This is equivalent to executing a PRINTER ON statement at the start of the program.
- S suppress the display of any non-fatal warning messages.

Figure 16.41: RUN options

If a program has been catalogued, these same options may be specified on the command which invokes the program:

```
ADD.STOCK (D
DELETE-RECORD (P
```

It is, of course, inappropriate to attempt to execute an external subroutine. If you attempt this, it will be rejected with a message of the form:

```
'XXXX' is a subroutine, not accessible from TCL.
```

The RUN-LIST command will execute, one after another, a sequence of programs submitted as a *select-list*, in a sequence such as this:

```
SSELECT WAGES.PROGRAMS = "YEAR.END]"
RUN-LIST WAGES.PROGRAMS
```

16.15.4 CATALOG command

The CATALOG command creates an item on the MD which points to the location of the object code for a program or an external subroutine. The format of the command is:

```
CATALOG file.name item.list
```

Some examples are:

```
CATALOG MBPROGS SENTENCE01
```

```
CATALOG STOCK.BP ADD.STOCK AMEND.STOCK DELETE-STOCK
```

```
CATALOG RU.PROGS *
```

<== Catalog all programs

```
CATALOG RU.PROGS
```

<== Catalog all programs

```
SSELECT PROGS = "[AMEND]"
```

```
CATALOG PROGS
```

There is only one option with this verb:

CATALOG MBPROGS SENTENCE01 (O

and this will overwrite any *existing* item called SENTENCE01 (unless it is a file definition) on the MD.

Figure 16.42 shows the format of the item which is set up on the MD when a program or a subroutine is catalogued. This holds codes which identify it as a catalogued pointer, and – most importantly – the name of the file and the program. This implies that we do not need to recatalog a program or subroutine each time we recompile it, but only if the name of the file or the program item changes. You might also note that the item-id of the catalogued pointer need not be the same as the program (or subroutine) to which it points (that is, the item-id need not be the same as the item name on line 4). However, if you want the catalogued pointer item to have a name other than that of the program or subroutine, you must modify the item yourself.

SENTENCE01	
01	VR
02	319
03	F
04	MBPROGS SENTENCE01

Fig 16.42: Catalogued pointer

It is not necessary for you to catalogue a *program*. If we were to catalogue the SENTENCE01 program discussed here:

CATALOG MBPROGS SENTENCE01

then our users would be able to execute the program by the RUN command, as before, or they could simply type the *name* of the program:

SENTENCE01

In contrast, an **external subroutine** must be catalogued so that it can be located when needed by the Basic CALL statement.

You can delete the object code for a program or a subroutine by a command such as:

DELETE DICT MBPROGS SENTENCE01

and you can delete the object code and the catalogued pointer on the MD by means of a command of the form:

DECATALOG MBPROGS SENTENCE01

16.16 Flash/Basic

When a Basic program is compiled, the **object program** is made up of a sequence of statements in the Pick **pseudo-object code**, and at run-time this code is **interpreted**, statement by statement, into the **machine code** for the particular machine which is being used. This is illustrated in Figure 16.43.

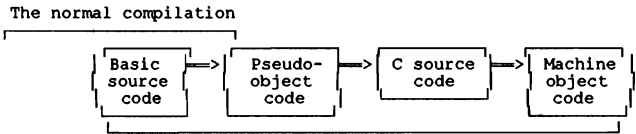


Figure 16.43: Basic or Flash/Basic

On AP/Unix implementations, an extended version of Basic object code is available. This is known as **Flash/Basic** and, as Figure 16.43 shows, the compilation differs from that of regular Pick/Basic:

- The normal Basic **source code** is first converted into the Pick **pseudo-object code**, as with any other compilation.
- This **pseudo-object code** is then converted into source code in the equivalent **C language source code**.
- The **C source code** is compiled directly into the **machine code** for the actual machine which is being used.

Using this machine code program is significantly faster than the standard Basic pseudo-object code.

These actions are invoked by including the **O** (for optimisation) option on the **BASIC / COMPILE** command. This option has the form:

O(n)

and produces **optimised** (Flash/Basic) object code at one of a number of **optimisation levels**, according to the value of *n*:

- 1 No optimisation.
- 2 Place the subroutines inline.
- 3 Optimise the C code using a minimum of memory.
- 4 Place subroutines inline and optimise the C code to produce the fastest code (combining levels 2 and 3).

Several options – **B / F / H / K / R / W / Y** – are available in conjunction with the **O** option on the **BASIC** and **COMPILE** commands when using Flash/Basic. Full details of Flash/Basic and of these options can be found in EPick.

If a program has been **flushed**, that is, compiled with the **O** option, then *all* the external subroutines which that program calls must also have been compiled in the same manner. However, if the program has not been flushed, then it will use the slower Pick pseudo-object code versions of any external subroutines which are called, even if flushed versions of the subroutines are available.

16.17 Summary

Most data processing on the Advanced Pick system is carried out by programs written in the Basic language. AP Basic has many extensions to exploit the features of Advanced Pick.

Each Basic source program is created as a separate item on a file. The source program is compiled by means of the **BASIC** or the **COMPILE**

verb; this reports any syntax errors in the program or produces the object program. The object program is executed by the RUN command; this invokes the run-time processor which interprets the Advanced Pick object code into machine code.

A program may be constructed as a main program and a set of internal and/or external subroutines. External subroutines are created, compiled and catalogued as separate program items.

Flash/Basic is available on AP/Unix systems and extends the compilation phase to the production of a machine code version of the program for faster execution.

17

Procs

Just as a *program* is a processing sequence written in the Basic language, so a **Proc** is a processing sequence written in the Pick **Proc language**. The Proc language was an important component of the original Pick system and allowed you to create and save an item which would invoke a TCL command or an Access sentence, or a sequence of commands and sentences, simply by typing in the name of the item. On Advanced Pick, Procs have largely been superseded by *macros*, *menus* and Basic programs, but they still have a place when it is required to invoke a sequence of commands and sentences which are to be tailored according to the user's requirements.

For example, if we wanted to have a simple means of allowing the user to issue the Access sentence:

```
    SORT STOCK WITH TYPE "DESK" AND WITH PRICE > "50"
```

we might create either the macro or the Proc shown in Figure 17.1, and either of these would do the trick.

```
DESK.50.M
```

```
M
```

```
SORT STOCK WITH TYPE "DESK" AND WITH PRICE > "50"
```

(a) Macro

```
DESK.50.PQ
```

```
PQ
```

```
HSORT STOCK WITH TYPE "DESK" AND WITH PRICE > "50"
```

```
P
```

(b) Proc

Figure 17.1: Saving an Access sentence

These items would be created on the MD, and whenever the user wished to issue the Access sentence, he would type in the TCL command:

```
DESK.50.M
```

to invoke the macro, or:

```
DESK.50.PQ
```

to invoke the Proc, and the required sentence would be executed and the report produced.

```
PQ
HSORT STOCK WITH TYPE
O
OWhat product type do you want+
IP?
A"
H AND WITH PRICE >
O
OWhat price do you want+
IP?
A"
P
```

(a) Version 1

```
:PROC.T.P

What product type do you want?TABLE

What price do you want?80
```

(b)


```
PQ
HSORT STOCK WITH TYPE
A"2
H AND WITH PRICE >
A"3
P
```

(c) Version 2

Figure 17.2: A Proc

But what if we wanted some means of letting the user type in just the product type and the price which he wanted, perhaps producing the sentence:

```
SORT STOCK WITH TYPE "DESK" AND WITH PRICE > "50"
```

on one occasion, and the sentence:

```
SORT STOCK WITH TYPE "CHAIR" AND WITH PRICE > "100"
```

on another? Could we do this? The answer is YES, with a Proc like that in Figure 17.2 (a). Obviously, this is slightly more complicated than the Proc shown earlier, but it is much more powerful. When this Proc is saved on the MD (I saved it as item PROC.T.P), and invoked, the display might look like that in Figure 17.2 (b), and this particular instance would issue the Access sentence:

```
SORT STOCK WITH TYPE "TABLE" AND WITH PRICE > "80"
```

If I had created the alternative version in Figure 17.2 (c) as MD item PROC.T.P.2, this would allow the user to invoke the Proc by means of a command such as:

```
PROC.T.P.2 TABLE 80
```

and would pick up the data from this command without any further user intervention. Obviously, we could write a Basic program to achieve the same results, but a Proc is fairly quick and simple.

All your new Advanced Pick application systems will probably be written using Basic programs, macros and menus, so there is no real need for you to master Procs. But it is quite possible that you are working on a system which has migrated from R83 Pick and which makes considerable

use of Procs. You may have to modify these Procs or even rewrite them as macros and Basic programs. It is for such people that I include this Chapter. I shall not go very deeply into Procs, but I shall show how to use the Proc language to perform some useful tasks.

17.1 Simple Procs

We can illustrate the general form and nature of Procs by some examples. Those starting with a PQ statement are complete Procs, the others are fragments.

A Proc to invoke a single command

PQ
HLIST STOCK WITH TYPE "DESK" AND WITH PRICE > "50"
P

A Proc to invoke a long command

PQ
HSORT STOCK WITH TYPE "DESK" "CHAIR" "SETTEE"
H BY PRICE
H BREAK-ON PRICE " 'U'"
H DESCRIPTION TOTAL VALUE
H HEADING "'C'* Price report for 'F' file *'LL'"
H LPTR
P

Several commands in one Proc

PQ
HSET-FLOPPY (AS
P
HT-REW
P
HT-DUMP STOCK
P
HT-REW
P

A Proc to execute a Basic program

PQ
HRUN MYPROGS CALC001
P

A Proc to copy from one file to another

PQ
HCOPY STOCK
STON
H(STOCK2
P

A Proc to select & copy from one file to another

PQ
HSELECT STOCK WITH QUANTITY = "0"
STON
HCOPY STOCK<
H(STOCK2
P

A Proc to sleep before releasing all spooler jobs

PQ

```

HSLEEP 22:00
P
HSP-EDIT MUS
P

```

A Proc to sleep before releasing a specific job

```

PQ
HSLEEP 23:50                                <== Sleep until late evening
P
HSP-EDIT 11
STON
HS<
HY
P

```

Displaying a message

```

OThis Proc produces the weekly statistics report
O
OPress <Return> to continue:+

```

Asking the user for a response

```

O
ODO you want to continue+
IP?

```

Vetting the user's response: 1

```

10 O
ODO you want to continue? Y/N+
IP?
IF A = Y GO 20
IF A = N GO 30
GO 10

```

Vetting the user's response: 2

```

10 O
OPlease enter the date as dd/mm/yy+
IP:
IF A = (1NON/1NON/1NON) GO 20
IF A = END XJOB TERMINATED
GO 10

```

Vetting the user's response: 3 - Adding LPTR

```

10 O
OSend report to Screen or Printer? S/P+
IP?
IF A = S GO 30
IF A = P GO 20
GO 10
20 C Add LPTR if required -----
H LPTR

```

30 C Not LPTR -----

Using the user's response in the output command

PQ

C Build up part of Access sentence -----

H SORT STOCK WITH MDATE >=

C Get start date -----

10 O

OPlease enter the start date in the form dd/mm/yy+

IP:

IF A = END XJOB TERMINATED

IF A # (1NON/1NON/1NON) GO 10

C Enclose date in " and add to sentence -----

A"

C Add next part of Access sentence -----

H AND WITH MDATE <=

C Get end date -----

20 O

OPlease enter the end date in the form dd/mm/yy+

IP:

IF A = END XJOB TERMINATED

IF A # (1NON/1NON/1NON) GO 20

C Enclose date in " and add to sentence -----

A"

C Add any more bits to sentence -----

H BY PRICE DESCRIPTION BREAK-ON PRICE

C Get destination = Screen or Printer -----

30 O

OSend report to Screen or Printer? S/P+

IP?

IF A = S GO 50

IF A = P GO 40

GO 30

40 C Add LPTR if required -----

H LPTR

50 C Process the sentence -----

P

Creating and saving a list

PQ

HSELECT STOCK WITH TYPE "DESK"

STON

HSAVE-LIST DESK.ITEMS

P

Recalling and using a list

PQ

HGET-LIST DESK.ITEMS

STON

H SORT STOCK BY PRICE DESCRIPTION BREAK-ON PRICE

H LPTR

P

Recalling and using a list to build another list: 1

```
PQ
HGET-LIST DESK.ITEMS
STON
HSELECT STOCK WITH PRICE <= "50"<
HSAVE-LIST DESKS.LESS.THAN.50
P
```

Recalling and using a list to build another list: 2

```
PQ
HGET-LIST OLD.INVOICES
STON
HSELECT INVOICES WITH FLAG = "1"<
HSAVE-LIST OLD.INVOICES
P
```

Repeating a task several times

```
PQ
1 C Get file name -----
IF A2 GO 2
O
OEnter the file name+
S2
IP:
IF A2 = END X
GO 1
2 C -----
S3
C To start at Modulo 11
```

```
IH11
20 HHASH-TEST
A2
STON
A3
P
S3
+2
GO 20
```

<== Hold modulo counter in word 3

<== Add 2 to the modulo counter

A Proc to invoke a Proc held on another file

```
PQ
(file.name item.name)
```

These examples illustrate a number of general points about Procs:

- Each Proc is held as an item on the MD. The above examples show the format.
- Each Proc is executed (or invoked) simply by typing the item-id of the Proc item on the MD.

- Each Proc statement is held on a separate line (or attribute) of the Proc item.
- The first statement is PQ
- Comments start with the letter C.
- The O statement displays a message on the screen.
- The general form of the simpler Procs is a sequence of H statements to construct the required command, followed by a P statement to process the command. Each P statement processes and then clears the previous command.
- The STON statement is used to construct a stack of data responses which feed the main command.
- The IP statement invites the user to type in data at the keyboard.
- The references A and A1 refer to a word of the data which the user typed in earlier.
- The A statement incorporates the user's response into the command which is being constructed. The A" statement encloses the user's response in " marks and then incorporates this into the command.
- Any statement may have a numeric label at the front. This is used as the destination of GO statements.

The statements which are available in the Proc language are shown in the Appendix. We shall look at some of these more closely in a moment.

17.2 Buffers and stacks

Important features of any Proc are the buffers which hold the data being processed. The organisation of the data within a Proc is illustrated in Figure 17.3. Here, you see the two **input buffers** and the two **output buffers**.

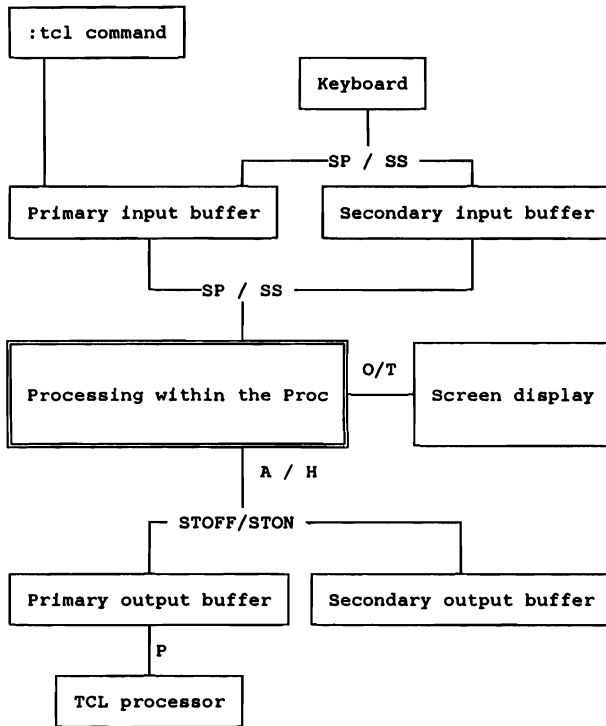


Figure 17.3: The Proc buffers

The TCL command which invoked the Proc is placed in the **primary input buffer**, and the individual words there can be handled by using references such as A1, A2 and so on. The IP statement can be used to accept data from the keyboard and place this in the input buffer. Data may be read into and picked up from either input buffer; the SP and SS statements serve as a switch to denote which input buffer is currently being used.

The O statement and the T statement are used to display text, messages and other information on the screen.

The H statement is used to place specific text into the output buffer, and the A statements can be used to move data from the input buffer to the output buffer. The STON and STOFF statements serve as a switch to denote which output buffer the data is to be sent to.

The P statement executes the command which has been constructed in the primary output buffer, and if this command requires further information, this will be picked up from the secondary output buffer (also known as the **stack**).

Many of the Proc statements are concerned with moving the data along these paths.

17.3 Handling the buffers

For anyone new to Procs, the most confusing concept is that of the **buffers**. As we have seen, there are two input buffers and two output buffers. All these buffers are simply storage areas holding strings of data, and it is convenient to think of these as strings of **words** or **elements** separated by spaces.

At any one moment, only one input buffer and one output are active. When you start to execute a Proc, the primary input buffer and the primary output are the active buffers. There are four statements to make each of the buffers active:

SS to set the secondary input buffer to be the active input buffer.

SP to set the primary input buffer to be the active input buffer.

STON to set the secondary output buffer (the stack) to be the active output buffer.

STOFF to set the primary output buffer to be the active output buffer.

In order to control the way in which the contents of the buffers are handled, the Proc processor uses an **indicator** to point to a particular element. This points to the *current* element in the *active* buffer. Some Proc statements move this indicator, others do not. You can use a statement such as:

S3

to move the indicator to element 3 of the active input buffer.

When you start to execute a Proc, the primary input buffer contains the TCL command which you typed to invoke the Proc. Thus, if you invoked a Proc called PROC.C.P by typing the TCL command:

```
PROC.C.P AAA BBB CCC
```

then the primary input buffer would contain the four elements:

```
PROC.C.P AAA BBB CCC
```

and the indicator would point to the first element:

```
PROC.C.P
```

which is the **current** element of the input buffer; the secondary input

buffer and both output buffers would be empty.

You can use the D0 statement to display the entire contents of the active input buffer, D to display the current element of the active input buffer, D4 to display the fourth element of the active input buffer, and so on.

If you need to ask the user for a piece of data as the Proc is executing, the IP statement will accept data from the keyboard and place this directly into the active input buffer, and the IS statement will accept data into the secondary input buffer. In either case, the data will be entered at the position indicated by the indicator for that buffer.

Having used the appropriate SS / SP / STON / STOFF statements to determine which are the active buffers, you can then move (or, to be precise, copy) data from the active input buffer to the active output buffer by means of statements such as:

- A moves the current element from the input buffer to the output buffer, enclosing it in spaces.
- A\ moves the current element exactly as it is.
- A' moves the current element and encloses it in ' characters.
- A" moves the current element and encloses it in " characters.
- A9 moves element 9 and encloses it in spaces.
- A\2 moves element 2 exactly as it is.
- A'3 moves element 3 and encloses it in ' characters.
- A"1 moves element 1 and encloses it in " characters.

When moving data to the **secondary output buffer**, all these forms move the data exactly as it is, and do not enclose it in any character.

The H statement is used to move a specific string of characters to the current element of the active output buffer, and the IH statement is used to move a specific string of characters to the current element of the active input buffer.

The P statement will process the command which has been constructed in the primary output buffer. If this command requires further information – as does the COPY verb which asks you where the data is to be copied to – then the contents of the secondary output buffer (the stack) will be used to feed this information. Having processed the command, the processing will clear the output buffers, reset their indicators, and return to the Proc for any further Proc statements.

17.4 P statements

In my examples, I have used the simple P statement to process the command which I have constructed in the output buffer. There are other forms of the P statement which are of value in certain situations. They all execute the command which has been constructed in the primary output buffer, but they have additional facilities:

- PH** is identical to the P statement except that any screen display produced by the executing command is suppressed. This might be useful if you were doing some action which produced output which might be confusing or of no interest to your users or which you wanted to hide from the users.
- PP** is identical to the P statement except that the PP statement will display the contents of the output buffers, showing the command which is about to be executed.
- PW** is like the PP statement except that the PW statement will display the contents of the output buffers, and then wait for the user to enter one of:
 - G** or <Return> to process the command and then continue with the Proc.
 - S** to skip the command which is about to be processed, clear the output buffers and then continue with the Proc.
 - N** or x to abandon the command and the Proc.

The PW command is useful when you are testing a Proc.

- PX** executes the command in the output buffer and then abandons the Proc, ignoring any further statements in the Proc.

17.5 Output: Displaying information

The O statement is used to display messages to the users, possibly to inform them of what the Proc is doing, or as a prompt when the user is expected to enter data for processing by the Proc. The format of the statement is:

`OEnd of Job`

to display the message `End of Job`, or just:

`O`

to output a blank line.

The statements D, D0, D1 and so on, will display the contents of the *input* buffer, as we saw earlier, and you can mix these with O statements. So the sequence:

```

OElement 1 contains +
D1+
O and 2 contains +
D2

```

might display:

```

Element 1 contains PROC.99 and 2 contains ABC

```

This example illustrates that, if a D statement or an O statement ends with the + sign, then this will hold the cursor so that the next displayed output will begin at that position.

We have seen that the PP and the PW statements can be used to display the contents of the output buffer just before they are processed. There is no other way of looking at the contents of the output buffers.

There is a more powerful statement for displaying data on the screen: the T statement. This has the general form:

```

T element1,element2,element3, ... elementn

```

comprising a sequence of elements separated by commas, such as:

```

T C,(0,15),"Please wait",B

```

The T statement holds the cursor after the last character which was displayed.

Some of the more useful elements are:

"xx" to display the string xx.

B to sound the bell on the terminal.

C to clear the screen.

(x,y) to move the cursor to column x of line y of the screen and print whatever follows starting in that position.

(-1) to clear the screen – exactly like the C parameter.

(-2) to position the cursor at the top left corner of the screen (the *home* position).

(-3) to clear the rest of the screen from the current cursor position.

(-4) to clear the rest of the current line from the current cursor position.

The forms (x,y) and (-n) are equivalent to the g(x,y) and g(-n) functions which are used with the Basic CRT and PRINT statements, which we met in Chapter 16.

Here are some more examples of the T statement in action:

```
T C,"** Produce the Stock Inventory List **"
T C,"** DISPLAY THE PERSONNEL LIST **",B
T B,(0,5),"The report is now being produced"
```

A long T statement may be continued on to further lines, if necessary, by ending each incomplete line with a comma. As this example illustrates, the continuation lines do not begin with a T:

```
T C,(25,0),"** General stock update routine **",
(5,5),"Reference:",(25),"_____",
(5,7),"Description:",(25),"_____",
(5,9),"Price:",(25),"_____"
```

Advanced Pick Procs have no direct means of outputting to the printer.

17.6 Control statements: GO / IF / X

In addition to manipulating data, the Proc language offers facilities which are normally provided with a programming language. We shall look at the GO statement, the X statement, and the IF statement.

The G or GO or GOTO statement, for example:

```
GO 123
```

jumps to the statement carrying the numeric label 123. You might use this to jump back to the beginning of a Proc and start again. There must be a statement with the label – 123 in this instance.

A statement such as:

```
XThis is the end of the job
```

or the equivalent sequence:

```
OThis is the end of the job
X
```

will display a message and then terminate the Proc.

The IF statement is used to test for a certain condition and take some course of action if the result is true. One form of the IF statement is:

```
IF A = value statement
```

for example:

```
IF A = YES GO 100
```

and this will test the current element of the input buffer (identified by A) and if this is just the word YES, then the statement GO 100 will be executed. If the current element is not YES, then no action will be taken

and the processing will continue with the statement following this IF statement. Some other acceptable forms are:

IF A2 = YES GO 10

will jump to statement 10 if the second element in the active input buffer is the word YES.

IF A = X XEnd of job

will check the current element in the active input buffer and, if this is the letter X, will display the message and terminate the Proc.

IF A1 = YES IF A2 = PLEASE XYOU SAID YES PLEASE

will check the first and the second element in the active input buffer and react accordingly.

You can use other operators where these examples use the = sign:

- # not equal to
- > greater than
- < less than
-] greater than or equal to
- [less than or equal to.

IF A # Y IF A # N XYOU MUST ENTER Y OR N

will check the current element of the active input buffer and display the message if this is neither Y nor N.

Other forms of the IF statement allow you to check that the data matches a certain **pattern**:

IF A = (3A) GO 20

will jump to statement 20 if the data in A is just 3 letters, such as ABC and XYZ and AAA.

IF A1 = (3A2N) GO 30

will jump to statement 30 if the data in A1 is just 3 letters followed by 2 digits: AAA11 or ABC92, but not AAAAA or 99999 or AA999.

IF A2 = (0N) GO 40

will jump to statement 40 if the data in A2 is only numeric digits (with no letters or other characters).

IF A3 # (1NON) GO 60

will jump to statement 60 if the data in A3 does not consist of 1 numeric digit followed by any number of additional digits.

These are the same patterns that we saw when we looked at the P processing code in Chapter 14, and the IF MATCHES statements in Chapter 16.

17.7 Subroutines

Suppose that we were writing a Proc which needed to perform the same processing – say, clearing the screen and displaying a title – from several places within the Proc. We could do this by constructing the Proc as in Figure 17.4.

```
PQ
OThis Proc performs the daily updates
GOSUB 100
HRUN PROGS CALC01
P
GOSUB 100
HRUN PROGS CALC42
P
:
GOSUB 100
HRUN PROGS CALC999
P
X
100 C Clear the screen and display the title
T C,(0,15),"** The daily update run **"
X
```

Figure 17.4: A Proc subroutine – internal

The statement GOSUB 100 will jump to statement 100 (where we clear the screen and display the title) and continue from there, and then the X statement will take us back to the statement following the GOSUB 100 statement. By doing this, we only have to write the T statement once but we can use it whenever we need to. We refer to a group of statements such as this as a **subroutine**, or, because it is held within the Proc which uses it, an **internal subroutine**.

Just as we needed to use the same T statement several times in our Proc, there may be occasions when you can use the same group of statements in several different Procs. In this case, you would create the statements as a separate Proc and jump to this Proc by means of a statement such as:

[MYPROCS PART9]

which will pass control to the Proc PART9 which is held on the file MYPROCS. When an X statement is encountered in PART9, control will be passed back to the original Proc. We refer to this process as **calling** another Proc, and the second Proc is known as an **external subroutine**.

Figure 17.5 shows how we might reorganise the processing of Figure 17.4 to use an external subroutine.

```
PQ
OThis Proc performs the daily updates
[MYPROCS DISP.TITLE]
HRUN PROGS CALC01
P
[MYPROCS DISP.TITLE]
HRUN PROGS CALC42
P
:
[MYPROCS DISP.TITLE]
HRUN PROGS CALC999
P
X
```

(a) The main Proc

```
PQ
C Clear the screen and display the title
T C,(0,15),"** The daily update run **"
X
```

(b) The external subroutine: DISP.TITLE on the file MYPROCS

Figure 17.5: A Proc subroutine – external

If you use a statement such as:


```
(MYPROCS FINAL.ROUTINE)
```

this will pass control to the Proc FINAL.ROUTINE which is held on the file MYPROCS, but control will not be passed back to the original Proc. We refer to this process as **chaining** to another Proc.

In all cases, the same input/output buffers are used by and accessible to all these Procs, so you don't have to concern yourself about passing data to and fro as you jump from Proc to Proc.

17.8 Arithmetic in Procs

The arithmetic capabilities of Procs are fairly restricted, allowing you only to add or subtract integer constants to the current element in the active input buffer. For example, the statement:

```
+2
```

will add 2 to the current element of the active input buffer, and:

```
-8
```

will subtract 8 from the current element of the input buffer.

You may find it necessary to use the Sn statement to position the indicator to the correct element.

17.9 File handling

The user-exit U01AD will allow you to read data from a file, and it is used in a three-line sequence:

```
U01AD
file.name item.id attribute code
error handling statement
```

where *file.name*, *item.id* and *attribute* specify the data which is to be read from the file, and *code* specifies the action to be taken. The *code* will be any one of:

- A output to the non-active output buffer.
- P output to the primary input buffer.
- S output to the primary output buffer.
- T display on the screen.
- V verify that the file exists.
- VA verify that there is a value in the attribute.

The *error handling* statement specifies the action to be taken if the file read statement does not execute successfully, that is, if the file does not

exist, if the item cannot be found or if the data attribute is null, according to the code. If the statement executes successfully, then the error handling statement will be skipped. Some examples are shown in Figure 17.6.

If any of the file name, the item-id, or the attribute number are to be picked up from the *input buffer*, then references of the form %*n* can be used to identify element *n* of the input buffer, as illustrated in Figure 17.6, and #*n* to identify element *n* of the output buffer.

```
U01AD
STOCK 1000 1 T
XCANNOT FIND THE FILE/ITEM
OItem found
```

(a)

```
PQ
1000 C
OEnter item-id+
IP
ORreading attribute 1: +
U01AD
STOCK %1 1 T
XCannot find the file/item
GO 1000
```

(b) Using the U01AD user-exit

```
Enter item-id: 1000
Reading attribute 1: DESK, GREEN-BLUE, ASH
Enter item-id: 9000
Reading attribute 1: SIDEBORD, YELLOW, ASH
Enter item-id: Z
Reading attribute 1: Cannot find the file/item
```

(c) Output

Figure 17.6: User-exit U01AD

User-exits U11AD and U21AD are also available file handling. Details of these can be found in EPick.

17.10 Summary

The Proc language is a means of constructing a sequence of TCL commands and/or Access sentences which can be invoked by typing the name under which they have been saved.

The Proc language has facilities for constructing TCL commands, for displaying messages on the screen, for accepting data from the user, for moving and testing data, and for performing simple arithmetic and simple file-read operations.

On Advanced Pick, simple Procs have been superseded by macros, menus and Basic programs, but users who have migrated from R83 Pick systems may need to be able to modify and understand existing Procs from their old system where Procs were widely used.

18

The debugging software

If an error occurs during the execution of a Basic program or any other process, Advanced Pick will pass control to one of two pieces of software: the **Basic debugger** or the **system debugger**. These are standard pieces of AP software and, in addition to indicating an error condition, they can be used by the programmer and the System Manager to locate, to identify and possibly to correct the error:

- The **Basic debugger** is invoked when certain fatal errors are detected – or the **DEBUG** statement is executed – during the execution of a Basic program. It can also be used by programmers as a tool for locating and identifying errors in a Basic program.
- The **system debugger** is automatically invoked when an error is detected during the execution of a system routine or a process written in the Pick Assembly language. It can also be used by programmers as a tool for locating and identifying errors in a Pick Assembly language routine.

Both debuggers are controlled by a series of commands typed in at the keyboard, and each command is carried out immediately, allowing you to continue the execution, to inspect areas of memory, to change the contents of areas of memory, to log off the system, and many other tasks.

18.1 Escape or debug?

If you are familiar with R83 Pick systems, you will know that the <Break> key was used to interrupt whatever was happening on your terminal, whilst the <Esc> key was only used in certain application programs, if at all. On Advanced Pick, the <Break> key and the <Esc> key play slightly different roles.

You will recall from the discussion of **level pushing** in Chapter 8 that Advanced Pick allows you to ...

press the <Break> key to suspend the current process, ...

push the processing to a lower level and then ...

issue a TCL command at that level.

When you push a level in this manner, the system replaces the normal system prompt by:

::

to indicate the level at which you are processing. You may then issue any TCL command to initiate a second process at this lower level. The second process may, in turn, be interrupted and a further operation invoked. When each process has completed at the lower level, the appropriate prompt:

::

will be displayed, and you may issue a further TCL command at this lower level. The following special TCL responses are available at this point:

- `<Return>` to *go up a level* and return to continue the execution which was interrupted.
- `END` to terminate the process which was interrupted. This action makes the level pushing feature similar to the action of the `<Break>` key under R83 Pick.
- `OFF` to log off the system.
- `DEBUG` to enter the Basic debugger (if you interrupted a Basic program), or the system debugger (if you interrupted a system process).

The roles of the `<Break>` key and the `<Esc>` key may be reassigned, and a number of TCL commands are available to control this: `BRK-LEVEL`, `ESC-LEVEL`, `BRK-DEBUG`, `ESC-DATA`, `ESC-TOGGLE`, as we discussed in Chapter 8.

In our discussions here, I shall use the symbol:

`<Interrupt>`

to denote the key which you press to interrupt a process. As we have seen, this may be the `<Break>` key (if `BRK-DEBUG` is active), or it may be `<Break>` followed by `DEBUG` (in the normal situation where the `BRK-LEVEL` is active).

18.2 Fundamental responses

Whenever the `<Interrupt>` key is pressed, the current process is suspended and the appropriate debugger is invoked. If a Basic program is interrupted, the Basic debugger is invoked and displays something like:

```
*I 123
*
```

showing that the Basic program was interrupted just as statement 123 was about to be executed. If any other process is interrupted, the system debugger is invoked and displays something like:

```
I cntrlnp:095
!
```

to indicate that the debugger has been invoked during the execution of, in this instance, a system routine identified by the name *cntrlnp*.

In either case, the appropriate debugger will wait for you to enter a command to indicate what you want to do. The two debuggers have slightly different commands. For example, a programmer using the Basic debugger might enter commands such as:

```
/VALUE
BS>100
```

and someone using the system debugger might enter commands such as:

```
C9999999,66;99
```

There are, however, a number of commands which are available with either debugger. The most important of these are:

```
G<Return>
^J
```

these both have the same effect, instructing the system to continue the processing, if possible.

```
END<Return>
END^J
```

to terminate the processing, leave the debugger and return to TCL or to the logon procedure for the account.

```
OFF<Return>
OFF^J
```

to abandon all the processing and log off the account.

```
P<Return>
```

to switch the terminal display ON or OFF, that is, disable/enable the display to the screen. The debugger messages are still displayed.

If your keyboard has been set up suitably, the <Line feed> key may be used instead of the ^J key. Used on its own, the <Return> key has no effect, except to repeat the debugger prompt.

The above commands are available with both the system debugger (at the ! prompt) and the Basic debugger (at the * prompt). If a user has system privileges SYS0 or SYS1, these are the only commands which he/she can use. Users with SYS2 privileges may use the other facilities and commands described below.

Let's look at the debuggers a little more closely, starting with the Basic debugger.

18.3 The Basic debugger

When you write a Basic program it may work the very first time you attempt to execute it. There will be times, however, when things do not go quite so smoothly: the program may not be producing the results which you expect, it may be going into an indefinite loop, it may terminate so quickly that you cannot see what happened. At such times you may feel the need for some means of watching the progress through the program, of slowing the program down, and of inspecting what values are held in which variables. So common is this need that most computer systems have a special piece of software – a **debugger** – which will help the programmer to locate the bugs (or errors) in a program.

A Basic programmer may be faced with several types of error in a program:

- 1) **Syntax errors:** these may be spelling mistakes and grammatical errors in the Basic source statements. They are reported by the compiler and usually prevent the program compiling successfully.
- 2) **Logical errors:** these may be spelling mistakes (you have used the name INCOME.TAX at one point, but ICOME.TAX at another), or errors in the calculation (you have added two numbers instead of multiplying them together) or errors in the logic (you have tested your data and then taken the wrong course of action).
- 3) **Fatal errors:** these are logical errors which prevent your program continuing (you have tried to store data in element 1000 of an array with only 100 elements).
- 4) **Non-fatal errors:** these are logical errors which are reported as the program executes, but from which the execution can recover (you have added a value into a TOTAL without first having set TOTAL to zero).

Syntax errors must be cleared up before you can produce an executable program, so the Basic debugger is mainly concerned with the detection of logical errors, fatal errors and non-fatal errors.

It is possible to debug a program without the use of the debugging software. For example, you might insert statements such as:

```
PRINT 'Now at line 400':; INPUT XXXX
PRINT 'Value of REC=':REC; INPUT XXXX
```

at strategic points within the source code and then watch the program

execution carefully. This works well (and many programmers prefer to do this rather than learn and use the elementary features of the Basic debugger), but it places an additional burden by requiring you to know where the problem lies, and by having to remember what and where your debugging statements were.

No matter whether or not you use the Basic debugger, it is good practice to design and construct your programs in such a way that it is relatively easy to locate the source of any errors in the processing and hence to correct the problem.

18.3.1 What the Basic debugger offers

In its major role of detecting fatal and/or non-fatal execution errors in your program, the Basic debugger will let you, the programmer:

- Interrupt the program execution at any point.
- Abandon the execution of the program.
- Go through the program statement by statement. You may also choose to proceed in steps of, say, 10 statements at a time.
- Redirect the processing to a particular statement or to the beginning of the program.
- Inspect the contents of any of the variables in the program.
- Change the contents of any of the variables in the program.
- Interrupt the execution when any of a number of conditions is satisfied.
- Specify a set of up to six variables which are to be displayed whenever an interruption occurs.
- Display the source statements of the program (or indeed any item on any file).
- Display the list of addresses in the subroutine RETURN stack.
- Pop addresses from the RETURN stack to return prematurely from a subroutine.
- Send the debugger output to the printer or to the screen.
- Pass to the system debugger.

This is a powerful set of tools which, when properly used, will enable you to identify and locate logical errors in a Basic program.

The Basic debugger may be used with standard Pick/Basic and also with Flash/Basic.

18.3.2 Invoking the Basic debugger

There are several ways in which the Basic debugger might be invoked as a Basic program is executing:

- 1) You can press the <Interrupt> key on your terminal keyboard, as we have already seen.
- 2) You can include the DEBUG statement in your program as a debugging aid to force an interrupt when a suspect piece of coding is about to be executed.
- 3) The Basic debugger is automatically called up when a fatal error is detected in the program.

In all cases, you will get a display something like:

```
*I40
*
```

showing that you are in the Basic debugger and the program has been interrupted as it is about to execute the statement on line 40 of the source program.

As we saw above, the commands:

```
END
OFF
P
G or ^J
```

may be issued at this point. If a user has system privileges SYS0 or SYS1, these are the only commands which are available. Users with SYS2 privileges may use any of the facilities and commands described below.

18.3.3 Symbol table and trace table

Before we look at the ways of using the Basic debugger, there are two entities which you must be aware of: the **symbol table** and the **trace table**.

The **symbol table** is a map which is produced by the Basic compiler and used by the Basic run-time processor, and which identifies the location of each variable by name.

```
01 * Demonstration program
02      DIMENSION STORE(10), STATS(4,5)
```

```

03      TITLE='HELLO'
04      PRINT TITLE
05      TEST=0
06 10   FOR X=1 TO 10
07          STORE(X)=X*X
08          PRINT X,STORE(X)
09      NEXT X
10      TEST=TEST+1
11      IF TEST=1 THEN GOTO 10
12      PRINT 'FINISHED'
13      END

```

(a) TESTP: source program

```

FID: 98765 : 0 0 0 0 ( 181CD : 0 0 0 0 )

000 :...h.....HELLO_...:
050 :..P.....D.....2.....:
100 :...V.....*_.....2.....UOP.....:
150 :.....=.....FINISHED_P..E_WORK^STORE^:
200 :000000320000000A000000001@.STATS^000000A00000000400:
250 :000005@.TITLE^0000000A00000000000000000000..TEST^00000:
300 :01400000000000000000..X^0000001E0000000000000000.._:
350 :.....:

```

(b) TESTP: object program and symbol table

Figure 18.1: Symbol table

Let us imagine that we have written a Basic program called TESTP. The source program is shown in Figure 18.1 (a). If we dump the frame(s) where the object program is held, we would see object code followed by the symbol table, as in (b). The names of the variables STORE, STATS, TITLE, TEST and X can be recognised in the symbol table. Following the name of each variable, we see the relative memory location where that variable is to be found and other information about the variable. When we use the Basic debugger, we can ask that the contents of, say, the variable TITLE be displayed; the Basic debugger will scan the symbol table for the entry for TITLE and use the information to find the actual memory address where the contents of TITLE are to be found.

The symbol table is generated by the compiler, and may be suppressed by the S option on the BASIC and COMPILE verbs. Suppressing the symbol table reduces the size of the object code, although this means

that you cannot use any debugger commands which refer to variable names.

The **trace table** is a list which is held and maintained by the Basic debugger, and contains:

- 1) The names of up to six trace variables whose contents are to be displayed each time the program execution is interrupted.
- 2) A set of up to four break conditions, the satisfaction of any one of which will invoke an interruption in the processing.

The trace table is empty when a program begins to execute. The contents are controlled by the user and are retained throughout the execution of a program (and its subroutines) and they are lost when the execution ends. Figure 18.2 shows the symbol table when we have issued .T commands:

```
TX
TSTORE(1)
```

to add two trace variables (as entries T1 and T2 in Figure 18.2) and a B command:

```
BX>5
```

to add a break-point (as entry B1). Having done this, the contents of the variables X and STORE(1) will be displayed whenever the program is interrupted, and the break-point indicates that we want to interrupt the program whenever the value of X is greater than 5.

As we shall see, there are other commands to remove such entries from the trace table.

T1	X
T2	STORE(1)
T3	
T4	
T5	
T6	
B1	X>5
B2	
B3	
B4	

Figure 18.2: Trace table

18.3.4 BASIC / COMPILE / RUN verbs

The BASIC and COMPILE verbs which compile your Basic program, and the RUN verb which executes the program, offer a number of options which allow you to make use of the Basic debugger when testing the program.

The following options on the BASIC (and the COMPILE) command are important if you intend to use the Basic debugger:

- c to produce an object program without **end-of-line characters**; these are markers in the object program which indicate the end of a physical line in the source program. Suppressing the end-of-line characters produces a smaller object program but any messages will refer to line 1. You should not use this option if you wish to use those facilities of the Basic debugger which use line-numbers.
- s to suppress the production of the **symbol table**. You should not use this option if you wish to use those facilities of the Basic debugger which use variable names.
- L to produce a source program listing as the program compiles.
- M to output a **program map** showing the names of the variables used, the statement labels used, and the distribution of the program statements across the frames occupied by the object code.

The Basic debugger can also be used with **Flash/Basic**.

Whenever you are testing a Basic program – with or without the aid of the Basic debugger – you should always be sure that you are using the correct version of the program and that the source program has not been changed since you last compiled it. You can waste a lot of time if you are looking at a printout of one version and executing another.

There are two run-time options which are important if you intend to use the Basic debugger. These are specified on the RUN command:

```
RUN file.name program.name {(options)}
```

or, if the program has been catalogued, on the command line which invokes the program:

```
program.name {(options)}
```

For example:

```
RUN SALES.PROGS ADD.RECORDS (D
ADD.RECORDS (D,E
```

The options are:

- D to invoke the Basic debugger at the first line of the program. This lets

you intercept the execution and call up the Basic debugger before you start.

- E to invoke the Basic debugger when any error condition is detected, no matter whether it is a **fatal error** or a **non-fatal error**.

You cannot invoke the debugger when you call a particular subroutine. You can, however, insert the **DEBUG** statement to interrupt the execution at a specific point in a program or a subroutine.

18.3.5 Basic debugger commands

Figure 18.3 shows some of the commands which are available for use with the Basic debugger. Some of these are provided specifically for use with Flash/Basic programs.

/	D	HELP	R
?	DE	I	RETURN
\$	DEBUG	K	S
#	DOWN	L	T
\	E	LP	UP
B	EDIT	N	Z
C	G	P	[

Figure 18.3: Basic debugger commands

Full details of the action and format of these commands can be found in EPick.

Now let's see how you will use the more important features to debug your programs.

18.3.6 Displaying / changing variables

The **/** command is used to display and change the contents of a specific variable. Thus, if you enter a command such as:

`/SURNAME`

the Basic debugger might respond with:

`SMITH=`

showing that the variable **SURNAME** currently contains the string:

`SMITH`

If you wish to change the value held in **SURNAME**, then you should enter the new value followed by **<Return>**. This new value will completely replace the previous contents of **SURNAME**. If you press **<Return>** without entering a new value, then the contents of **SURNAME** will be left unchanged.

Some other examples:

`/TITLE`

to display and allow you to change the contents of a variable `TITLE`.

`/STORE(7)`

to display and allow you to change the contents of element 7 of the dimensioned array `STORE`.

`/STORE`

to display – one by one – each element of the dimensioned array `STORE`.

`/STATS(2,3)`

to display element (2,3) of the dimensioned array `STATS`.

`/*`

to display the names and contents of *all* the variables (and all elements of any dimensioned arrays) in the program. You will not be allowed to change any of the displayed values.

If you attempt to inspect a variable which is not in the symbol table (or if the program has been compiled without producing a symbol table), then the debugger will display:

```
SYM NOT FND
```

that is, SYMBOL NOT FOUND.

If you attempt to display the contents of a variable which has not yet been assigned a value, then the debugger will display the messages:

```
O=
UNASGN VAR
```

that is, UNASSIGNED VARIABLE. It is possible to change the contents of a previously unassigned variable.

If you attempt to display the contents of a file-variable, then the debugger will display:

```
File-variable used where string expression expected
```

It is not possible to change the contents of a file-variable.

You can add up to six variables to the trace table, by means of commands such as:

`TTITLE`

to add the variable TITLE to the trace table, causing the contents of TITLE to be displayed at each break-point and each interrupt.

TSTORE(7)

to add element 7 of the dimensioned array STORE to the trace table.

TSTORE

to add element 1 of the dimensioned array STORE to the trace table.

The names and contents of all variables in the trace table will be displayed at each interrupt. You can remove entries from the trace table by means of the U command:

UTITLE

USTORE(7)

USTORE

Use the D command if you want to see the contents of the trace table.

18.3.7 Following the program flow

To follow the flow of your program, you must first invoke the Basic debugger by any of the means discussed, and then use a command such as:

E1

to proceed **one statement at a time**, interrupting the execution immediately before each statement is executed. The debugger will then display something like:

*E10

indicating that it is about to execute statement 10. To carry on with the execution, you should press:

^J

and the next statement will execute; repeated use of ^J will proceed with the execution statement by statement. If you want to inspect the source program to see what the actual Basic statement is, you can use the:

L

command to display the current statement, and the special forms:

L* to display the entire source code.

L5-17

to display lines 5 to 17 inclusive.

Other useful commands are:

G17 to jump to statement 17 and continue the execution from there.

E to cancel the step by step execution.

If you want the execution to interrupt when a certain condition is reached, you can do this by B commands of the form:

B\$=17

to interrupt when statement 17 is about to be executed.

BX>100

to interrupt when the variable X reaches a value greater than 100.

Each such **break-condition** is added to the trace table. You may have up to four break-conditions in action at any one time. Entries may be removed from the table by means of the K command:

K\$=17

KX>100

Use the D command if you want to see the contents of the trace table.

18.4 The system debugger

The system debugger is a very powerful tool, and – in the wrong hands or in inexperienced hands – it could be a very dangerous tool. Its main functions are:

- As a tool for debugging processes written in the Pick Assembly language,
- To inspect the contents of any frame of virtual memory,
- To change any part of any frame of virtual memory.

It is this last capability that poses a threat to the security and integrity of the system, for, by inadvertently changing the wrong frame, you can corrupt the standard processes and/or lose valuable data. The extended features of the system debugger are only available to users with SYS2 privileges; only the commands END / OFF / P / G are available to other users.

The system debugger should only be used by a fully experienced Assembly language programmer. For this reason, the system debugger is rarely – if ever – used in most organisations, and we shall only look at limited features of the system debugger here.

As we saw earlier, the system debugger can be invoked by pressing the <Interrupt> key, and you may enter any of the commands:

END
OFF
G
^J
P

If you invoke the system debugger accidentally, you can press the G<Return> or the ^J key to continue as if nothing had happened.

18.4.1 System debugger commands

The commands and responses described here should only be used by an experienced user who is aware of the potentially disastrous consequences of misusing the facilities of the system debugger.

Figure 18.4 shows the commands which are available for use with system debugger.

:	D	E	M	OFF	TIME
A	DB	END	MD	P	U
ADDD	DIVD	G	ME	R	X
ADDX	DIVX	I	MULD	SUBD	XTD
B	DRS	K	MULX	SUBX	Y
C	DTX	L	N	T	Z

Figure 18.4: System debugger commands

Full details of the action and format of these commands can be found in EPick.

18.4.2 Address and window

We know that the DUMP command can be used to see what's held in any particular frame. You can also use the system debugger to *inspect* the contents of a frame, or just a part of a frame, or even just a single character, and, more importantly, the system debugger will allow you to *change* the contents of that part of the frame; the System Manager might wish to do this as a last-ditch attempt to recover from a group format error.

In order to instruct the system debugger which part of virtual memory you are interested in, you will specify the appropriate **address**. All the standard Advanced Pick routines have identifying addresses such as:

```
bc.err.ptr
cntrlnp
date.fmt
ed.nl.top
fm.delete.dptr
get.list
```

but you will have no reason to inspect these. You may however wish to inspect the frames within one of your data files where, say, a **group format error** has occurred. The C and X commands will let you inspect a part of a frame of virtual memory. These have the form:

```
Caddress>window
Xaddress>window
```

There are other commands which will display parts of virtual memory, but we shall not discuss these here.

On the C and X commands, the address of the data within virtual memory is specified by means of two pieces of data:

```
fid.displacement
```

showing the **frame identifier** (the FID), and the **displacement** or **offset** of the address from the start of the frame.

The FID is expressed in a form such as:

12345 to identify frame 12345,

.3039 to identify frame 12345 (hexadecimal 3039 = decimal 12345). The full stop indicates that the FID is specified as a **hexadecimal** number.

The **displacement** is the number of bytes from the start of the frame at which the display is to start, and is expressed as a decimal number (preceded by a comma):

,12

or as a hexadecimal number (preceded by a full stop):

.C

Thus, to refer to the first byte (byte 0) of frame 12345 (decimal), the complete address might be specified in any of the forms:

C12345,0
C12345.0
C.3039.0
C.3039,0

and to refer to the thirteenth byte, that is byte 12, of frame 12345 (decimal), the complete address might be specified in any of the forms:

C12345,12
C12345.C
C.3039.C
C.3039,12

There are other features of the address specification, but we shall not discuss these here.

The third piece of information – the **window** – indicates the number of bytes which are to be displayed, and may be expressed as a decimal number:

10

or as a hexadecimal number:

.A

Thus, the commands:

C98765,0;10

would display 10 bytes starting at the first byte of frame 98765. Note that 0 is the displacement (or offset) of the first byte in the frame.

C12345,12;23

would display 23 bytes starting at byte 12 (the thirteenth byte) of frame 12345.

C61986,374;.58

to display the 88 bytes (88 decimal = 58 hexadecimal) starting at the 375th byte of frame 61986.

C33397.1AE;73

to display the 73 bytes start starting at the 431st byte (430 decimal = 1AE hexadecimal) of frame 33397.

C.3039.1AE;.58

to display the 88 bytes, starting at the byte 430 of frame 12345.

In each case the current data will be displayed and you would be able to:

- o Change the current data in the window.
- o Press <Return> to leave the data unchanged and return to the debugger prompt;
- o Press ^J to display the next window.

There are other features of the window specification, but we shall ignore these here.

18.4.3 Inspecting frames of virtual memory

The C and X commands of the system debugger are used to display the contents of virtual memory, C presenting the data in character format, X presenting the data as pairs of hexadecimal digits. As we have seen, these have forms such as:

C1234,56;10

to display the contents of the 10 bytes starting at byte 56 of frame 1234 in character format. Non-graphic characters will be shown as full stops.

X1234,56;10

to display the contents of the 10 bytes starting at byte 56 of frame 1234 in hexadecimal format.

In each case, the system debugger will display the current contents of the specified window of the frame in the required format (character with the C command or hexadecimal with the X command), and then invite you to enter the new contents of that window. For example, if we issue any of the above commands, the sequence might look like this:

```
!C12345,56;10 44-7FF^9C0=
```

```
!X12345,56;10 .34342D374646FE394330=
```

showing the contents of the 10 bytes in the respective formats. At the:

=

prompt, there are several possible courses of action: you may abandon the process, look at a further window or change the data in the window. You indicate your choice by one of the following commands:

<Return> to leave the data unchanged and return to the ! prompt.

^J to leave the data unchanged and display the next window.

- ^N to leave the data unchanged and pass on to the next window, displaying the address and the data there. The letters EOF indicate that the end of the frame has been reached.
- ^P to leave the data unchanged and then pass on to the previous window, displaying the address and the data there.
- A to redisplay the window.
- c to redisplay the window in character format.
- x to redisplay the window in hexadecimal format.

In the following section, we look briefly at how we might change the data in the window.

18.4.4 Changing frames of virtual memory

When a C or X command has been given and the appropriate data displayed, the system debugger will give you an opportunity to change that data:

```
C1234,56;7 12..34A=
```

showing that the seven bytes starting at the 57th byte of frame 1234 currently contain the characters:

```
12..34A
```

The characters represented by dots are most likely to be non-graphics or characters which cannot be displayed directly. If we had used the X command, we might have seen:

```
X1234,56;7 31320107333441=
```

showing the precise hexadecimal contents of the same seven bytes. If the contents of the window *are* to be changed, then the replacement data may be now be entered. The replacement data may be expressed in any of the following forms:

```
'cccc<Return>
```

will replace the data in the window by the characters ccccc. The apostrophe indicates that **character** data follows.

```
.xxxxxxxx
```

will replace the data in the window by the hexadecimal string xxxxxxxx – there must be an even number of hexadecimal digits. The full stop indicates that **hexadecimal** data follows.

If the replacement data consists entirely of the normal keyboard characters, then this may be conveniently expressed by means of the:

```
'cccccccccc
```

specification. If the correct data, which is to be entered, contains any special characters or non-keyboard characters, such as attribute-marks (hexadecimal FE) and segment marks (hexadecimal FF), then the replacement data is most conveniently expressed in hexadecimal notation by means of the:

.XXXXXXXXXX

specification.

In all these cases, the entire string of new data will be used to overwrite the original data, irrespective of the size of the window. Thus, if you specify a window of 10 but only enter five characters, then the last five characters in the window will be unchanged, and if you specify a window of 10 and enter 17 characters, then these will overwrite the 10 characters of the window and the 7 characters beyond the window.

18.4.5 Patching to recover from GFEs

When we discuss **group format errors**, we see that a last-resort and most desperate means of recovering data which has been lost as a result of a GFE is to use the system debugger to **patch** the offending frame, replacing the incorrect bytes by the correct data. We include this technique here in order to illustrate an application of the system debugger.

This activity should only be carried out by experienced technical users who are aware of the potentially disastrous consequences of making a mistake.

In order to patch a frame you must know – and you should write down – the following information:

- The FID of the frame which you will be patching. You can determine this by looking at the GFE-FID and the GFE-FILE fields of the items which logged the GFE on the ERRORS file, as we mention in Chapter 22.
- The exact location of the incorrect data. This is the **address** of the data and consists of the FID of the frame and the **displacement** of the start of the incorrect data from the beginning of the frame.
- The length of the incorrect data. This is the **window**.
- The incorrect data itself. This enables you to confirm that you have displayed (and will be changing) the string which you want to change.
- The correct data. Write down the exact characters which you want to put into the frame. If there are any non-keyboard characters, then you

should write the new data in hexadecimal format.

Armed with this information, you are now ready to patch the frame. The steps to be taken are as follows:

- 1) Use the TCL DUMP command to print a copy of the contents of the frame before you start to make any changes. For example, the command:

```
DUMP 12345 XP
```

will print the contents of frame 12345 in character format and in hexadecimal format. The hexadecimal format will reveal any characters which cannot be printed.

- 2) Enter the system debugger by typing the:

```
DEBUG
```

command at TCL.

- 3) When you get the ! prompt character, enter the display command in the form:

```
Cf,d;w  
or:  
Xf,d;w
```

where *f* is the FID of the frame which is to be patched; *d* is the displacement of the first byte in the string which is to be changed; *w* is the window (the length of the data string to be changed). If you use the C command, the data will be displayed as characters; if you use X, the data will be shown in hexadecimal.

Pay particular to the punctuation here. An invalid command will be rejected by the system debugger.

- 4) The system debugger will then display the required window. Check that this is the expected string.

If not, hit <Return> and try again from step (3), or go to step (7) and abandon the task.

- 5) Type the correct data as a series of display characters (preceded by an apostrophe) or as a series of *pairs* of hexadecimal characters (preceded by a full stop). For example:

```
'YES  
or:  
.594553
```

Don't press the <Return> key until you have checked the data.

- 6) When you are satisfied that you have typed the correct data, hit the <Return> key.

The frame has now been changed.

- 7) Repeat the same command as before:

```
cf,d:w  
or:  
xf,d:w
```

to confirm that the change has been made correctly.

If not, go to step (3) and repeat the procedure.

- 8) Abandon the system debugger by entering:

```
END
```

followed by the <Return> key.

- 9) DUMP a copy of the frame after you have made the changes and confirm that the action has taken place successfully. If not, repeat the process.

Remember that, as soon as you hit <Return> at step (6) the frame is changed. **There is no second chance.**

18.5 Summary

Pick has two pieces of software for locating and fixing programming and other errors: the Basic debugger and the system debugger. The Basic debugger is used to report, locate and fix errors within Basic programs. The system debugger is used with routines written in the Pick Assembly language.

The debuggers are called up by pressing the <Interrupt> key as a process is executing and they are invoked automatically when certain types of execution error are detected.

The debuggers are controlled by a sequence of commands entered at the keyboard. These commands allow you to perform a variety of actions: to inspect and/or change data being used by the process; to inspect the source code; to restart the process from a specific point or from the place where the interruption occurred. The debuggers will also let you terminate the process and log off.

19

Printing : the spooler

Because Advanced Pick is a multiuser system, there may be several people using the computer at the same time. At any one moment, any – or all – of these users may ask for a **printed report** by means of an Access sentence such as:

```
SORT STOCK BY PRICE DESCRIPTION TOTAL VALUE LPTR
```

which includes the LPTR modifier, or a TCL command such as:

```
COPY DICT STOCK (P
```

which uses the P option, or by executing a Basic program which uses the:

```
PRINTER ON
```

statement to print a report. If all the users were to ask for a report at the same time, there could be problems. Without some means of resolving the competition for the printer resources.

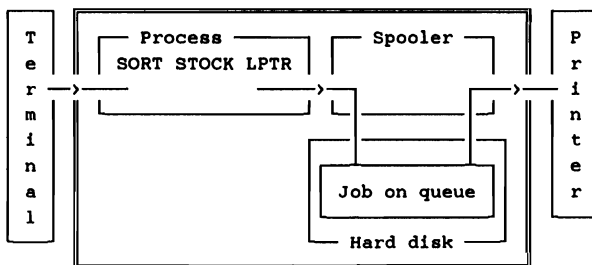


Figure 19.1: The spooler

The **spooler** is a standard part of the Advanced Pick system and controls the production and output of all the reports and other output which the users send to the printer. Whenever you perform a task which produces printed output, the spooler collects the output – line by line – and stores it on a temporary disk file until the process finishes. When the process is complete your terminal is released and is free to perform other tasks. In the meantime, the spooler takes care of the output, holding the various reports in a queue until the printer is free and your report can be printed. When the report has been output, the temporary disk file is deleted. This

is illustrated in Figure 19.1.

The process of collecting the output and saving it on disk is known as **spooling**, and that of sending the collected disk output to the printer **despooling**.

If there were no spooler, whenever you sent a report to the printer, you would have to wait until your entire job had finished and your report was completely printed before your terminal would be released and you could continue with other work. Moreover, if there was a long a queue of jobs to be printed, you might have to wait some time for your work.

19.1 The users, the form-queues and the printers

As you can see from Figure 19.1, each user can think of his/her terminal as being linked directly to the printer; as each printed report is requested, it seems to go straight to the printer – finally being printed when the report process has finished and the printer is free. There is, however, a little more going on behind the scenes. The diagram in Figure 19.2 shows the logical linkage between the users and the printer(s) which are available on the system.

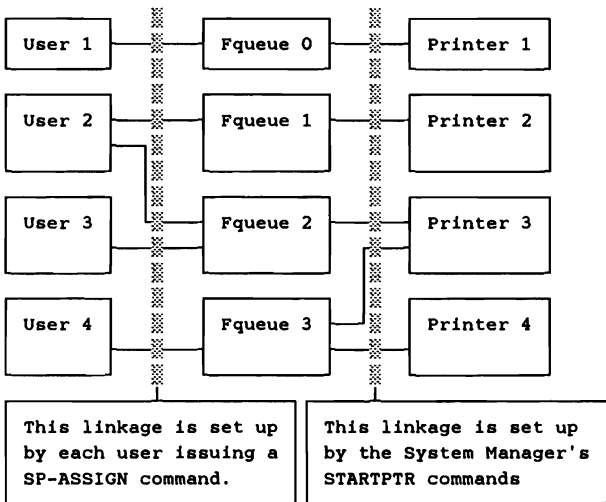


Figure 19.2: The form-queues

Between the users and the printers come the **form-queues**, the queues of reports which are currently being processed by the spooler. On your system, there may be just a single form-queue or there may be several. If there are several printers available on your system, each printer will normally be associated with its own form-queue and each user may assign his/her output to any of these queues – and thence to a particular

printer.

The first part of the linkage – the **spooling** stage – is that between the users and the form-queues:

- A user may send his/her print jobs to any queue (in our illustration, user 1 is linked to form-queue 0).
- A user may be linked to several queues at the same time (User 2 is linked to form-queue 1 and form-queue 2). He/she might be running a program which produces one report on a laser printer and another report on ordinary printout paper.
- Several users may be linked to the same queue (user 2 and user 3 are both linked to form-queue 2).
- This linkage between a user and the form-queue(s) is set up by the SP-ASSIGN command which is issued by the individual user.
- Unless they issue an SP-ASSIGN command to change their linkage, all users are connected to form-queue 0.

The second part of the linkage – the **despooling** stage – is between the form-queues and the printers:

- A form-queue may be linked to any printer (form-queue 0 is linked to printer 1).
- A form-queue may be linked to several printers (form-queue 3 is linked to Printer 3 and Printer 4). Any report sent to form-queue 3 will be printed on printer 3 or, if that is busy, on printer 4.
- A form-queue may be linked to a non-existent printer. This allows the users to produce reports and hold them until they are required; the jobs can be moved to a form-queue associated with a printer for output when required.
- Several form-queues may be linked to the same printer (form-queue 2 and form-queue 3 are both linked to printer 3).
- This system-wide linkage between the form-queues and the printers is set up by a set of STARTPTR commands issued by the System Manager.

The spooler can handle any number of users (up to the maximum for your system), any number of jobs (up to a maximum of 600 at any one time), any number of form-queues (up to a maximum of 126), and any number of printers (up to the maximum for your system).

19.2 When do you use the spooler?

If you only want:

- To print a single copy of your output,
- On the main printer (printer 0),
- As soon as the printer is free,

then you do not need to concern yourself with the spooler. You can simply carry out the work which produces the reports and these will be collected by the spooler and printed when your turn comes in the queue.

However, because of the sequence of events described earlier, the spooler can do much more than this:

- You may ask the spooler to suppress the output. You may be testing a program and wish to see whether the program runs properly before you concern yourself with the printout it produces.
- You may ask the spooler to hold the report on the form-queue and not print it. You may be producing a long report and don't want to tie up the printer for other users. You can then release the held report later and leave it printing overnight.
- You may specify that a sequence of several reports be collected together and printed as a single report.
- You may assign your report(s) to the form-queue associated with any of the printers which are available on your system.
- It is possible for a single program to produce several different reports: a payroll program might produce pay cheques, pay slips, departmental returns, and so on. In such cases, the spooler will allow you to specify that each of the reports be directed to a different printer, or some of the reports be directed to one printer, and other reports to another.
- You may ask for several copies of a report. This will avoid your having to run the same job several times in order to produce several copies.
- You may ask that your reports be dumped to backing storage diskette/tape for storage and printing when required.
- You may ask that the beginning of a report be printed first, allowing the operator to line up any special pre-printed stationery such as invoices or cheques.

When a report is held on one of the form-queues, you may:

- Inspect it – to check the results before printing.
- Delete it – the results may be wrong or it may be a test report.
- Move it to another form-queue – from where it will be output to another printer.

- Hold the report until you have had a chance to load and line-up the stationery on to the printer – you may want to print on cheques or special headed stationery.
- Copy the report and save it as an item on a file – you may want to incorporate an output report into some other documentation or save it for later use.

For these reasons, it is worth thinking about what the spooler can do for you.

19.3 Starting the spooler

The spooler is initiated automatically when the system coldstart is performed. Once the spooler has been started and the printers have been defined, the system normally requires little attention from the System Manager.

A number of verbs are provided for the System Manager to handle any abnormal situations, and to report on the status of the spooler and the printers, and to control the spooler:

ASSIGNFQ	SP-ASSIGN	SP-OPEN	STARTPTR
LISTABS	SP-CLOSE	SP-STATUS	STARTSPOOLER
LISTPEQS	SP-EDIT	SP-TAPEOUT	:STARTSPOOLER
LISTPTR	SP-KILL	START.SS	STOPPTR

Let's look at some of these:

START.SSs normally included in the **SYSTEM-COLDSTART** procedure and will start the **spooler** and the **phantom scheduler**.

STARTSPOOLER may be issued by the System Manager to re-initialise the spooler and resume normal operation, if the spooler has stopped for any abnormal reason.

STARTPTR will be used by the System Manager to define a printer, associate it with one or more form-queues and initialise it for use. Typically, the necessary **STARTPTR** commands will be included in the **SYSTEM-COLDSTART** procedure.

The general form of the command is:

```
STARTPTR n,q,s,tp,A {(options)}
```

where the various parameters are:

- n is a number identifying the logical printer which is to be started. The printer numbers can be chosen and allocated quite arbitrarily.
- q is the form-queue number(s) to be associated with the printer. If the printer is to be associated with more than one form-queues (up

to a maximum of three), then the form-queue numbers will be enclosed in parentheses and separated by commas:

(1,3,7)

The form-queue numbers can be chosen and allocated quite arbitrarily.

- s is the number of pages which are to be skipped between output reports. For reasons of economy, this will typically be 0.
- t identifies the type of printer and will be one of P for a parallel printer, or S for a serial printer. All AP/Unix printers are considered to be serial.
- p is the port number to which the printer is attached.
- A indicates that the form alignment process is to be executed, giving the user an opportunity to align the stationery correctly before each report is produced.

Some examples of the STARTPTR command:

STARTPTR 0,0,0,P0

to define the parallel printer on port 0 as printer 0, associate it with form-queue 0 and specify that no blank pages are to be skipped between output reports.

STARTPTR 1,2,0,S3

to define the serial printer on port 3 as printer 1, associate it with form-queue 2 and specify that no blank pages are to be skipped between output reports.

STARTPTR 2,(0,1,3),1,P4,A

to define the parallel printer on port 4 as printer 2, associate it with form-queues 0, 1 and 3, specify that a single blank page is to be skipped between output reports, and request the alignment process at the start of each output report.

STARTPTR 3,0,,,A

to reset the printer 3 for form-queue 0 and specify forms alignment.

STARTPTR 4

to restart printer 4 which has been stopped by a STOPPTR command or by the response T during the forms alignment process. The details of the printer remain unchanged.

Because of the wide range of features available on modern printers, the **ASSIGNFQ** command is required to associate a particular set of physical characteristics with a form-queue. The physical characteristics – such as those for printing the various fonts and faces – are set up by the

DEFINE-TERMINAL command and are held as items on the DEVICES file.

```
:LISTPTR
```

			PAGE	DEV OR	
TYPE	NUMBER	OUTPUT QUEUES	SKIP	LINE #	STATUS
PARALLEL	0	0	0	0	INACTIVE
SERIAL	1	2	0	3	INACTIVE
PARALLEL	2	0,1,3	1	4	INACTIVE

```
:SP-STATUS
```

THE SPOOLER IS INACTIVE.

PRINTER # 0 IS PARALLEL, INACTIVE, AND ON LINE.

THE PRINTER IS DEFINED AS PARALLEL PRINTER # 0.

ASSIGNED OUTPUT QUEUES: 0.

THE NUMBER OF INTER-JOB PAGES TO EJECT IS 0.

```
:LISTABS
```

LINE	STATUS	COP	FORM
#		IES	#
0	P	1	0
1	HS	1	0
2	P	2	1
3	HST	1	2
4	S	0	0
5	S	0	0

Figure 19.3: LISTPTR / SP-STATUS / LISTABS

LISTPTR and **SP-STATUS** are available to all users and will display details of the printers which are available and the form-queue(s) with which each is associated, as illustrated in Figure 19.3. The SP-STATUS output is also included in the WHERE report.

STOPPTR will be used by the System Manager to stop any of the printers and withdraw them from the system.

LISTABS is available to all users and will display the spooler assignments of all users, as illustrated in Figure 19.3.

19.4 SP-ASSIGN

We have seen that you will normally produce:

- o a single copy of each report;
- o to be printed;
- o on form-queue 0;
- o as soon as this is free.

These pieces of information are known as your **spooler assignments**. Anyone who wishes to exploit the facilities offered by the spooler must assign the appropriate spooler assignments. This is done by means of the **SP-ASSIGN** command.

The general form of the command is:

```
SP-ASSIGN {options}
```

Any number of options may be specified, in any combination and in any order. The most frequently-used options are:

- ? to display the spooler assignments.
- n to produce *n* copies of all subsequent reports. This may be any number 1 to 125; the default is one copy.
- F*n* to send the output to form-queue number *n*. This may be any number 0 to 125; the default is F0.
- H to hold all subsequent reports on the queue. Without this option, all reports will be deleted from the form-queue when they have been processed.
- s to suppress the output of all subsequent reports as they are completed. Without this option, all reports will be submitted for printing when they have been completed.

Some examples:

SP-ASSIGN
with no options will cancel all previous assignments and reset them to the defaults (one copy to be printed on form-queue 0).

SP-ASSIGN ?
to display your current spooler assignments, as illustrated in Figure 19.4, without changing them. The **LISTABS** command will display the

:SP-ASSIGN ?			
LINE	STATUS	COP	FORM
#		IES	#
0	P	1	0

Figure 19.4

spooler assignments of all users, as illustrated in Figure 19.3.

SP-ASSIGN 3

send three copies of all subsequent output to form-queue number 0 (the default form-queue).

SP-ASSIGN F2 3

send three copies of all subsequent output to form-queue number 2.

SP-ASSIGN F2 3 ?

this is identical to the previous command, except that the new spooler assignments will be displayed.

SP-ASSIGN S

suppress the output (the despooling) of all reports as they are completed. Since there is no H option to *hold* the reports, they will simply pass through the spooler without any output being produced or held. This is an unusual situation which might be used when testing a process without any concern for the output produced. Since no form-queue specification is included, the default (form-queue 0) will be used.

SP-ASSIGN H

send a single copy of the report to form-queue 0, and the report will be printed (because no S option is specified) and then held on the form-queue for further processing. See the SP-EDIT command below.

SP-ASSIGN H 5

print five copies of the report on form-queue 0, and the report will then be held on the form-queue for further processing.

SP-ASSIGN HS

SP-ASSIGN H S

SP-ASSIGN S H

These forms are equivalent and request that all output be *held* on the form-queue until you release it with the SP-EDIT verb. The S option will suppress the printing of any copies of the report.

SP-ASSIGN R3 F2

assign the output from the Basic PRINT ON 3 statement (and the output from the RUNOFF PFILE 3 command) to form queue number 2.

Less frequently encountered are the options:

- c to send the output to the printer immediately one frame of output has been collected by the spooler. See also the l option.

- I attach the printer to the user's process and print the report immediately without spooling to disk. See also the C option.
- O leave files open so that all subsequent reports will be output together as a single report. This is equivalent to the SP-OPEN command.
- T output to backing storage diskette/tape. The output may be recovered by means of the SP-TAPEOUT command.

Each SP-ASSIGN command – except SP-ASSIGN ? – cancels any previous spooler assignments.

19.5 SP-KILL

It may happen that you have sent a job to be printed and then changed your mind. The command:

SP-KILL

will cancel the job which is currently printing by terminating any further output to the printer for that job, and when the printer buffer is empty, the report will terminate with the message:

ABORT!

The full range of actions of SP-KILL includes:

- O terminate the report which is currently being output. If the report does not have hold status, then it will be lost.
- O convert to hold status a report which is about to be output, and
- O remove a printer from the spooler.

You must have SYS2 system privileges in order to kill reports other than those which you have sent to the spooler. Here are some examples:

SP-KILL

terminate the report which is currently being printed for the current user. The message ABORT! will be printed. If the current user's report is not being printed, then no action will be taken.

SP-KILL N

terminate the current report, as above, but suppress the ABORT message.

SP-KILL 'DM'

stop all spooler reports produced by user-id DM.

SP-KILL 2

terminate the report which is being output on printer number 2.

SP-KILL F3

convert report number 3 (that is, spooler entry number 3) to hold status. As each job is submitted to the spooler, the entry number is displayed. The LISTPEQS report also shows the report numbers, as in Figure 19.5.

SP-KILL F3-6

convert reports number 3, 4, 5 and 6 to hold status.

SP-KILL F4O

convert report number 4 – which is currently in the queue awaiting output – to hold status. The final character is the letter O.

SP-KILL F3-6O

convert reports 3, 4, 5 and 6 – which are currently in the queue awaiting output – to hold status.

19.6 LISTPEQS

Spooler Queue Elements.									
Job	Stat	Lnk	Line	Status	Copies	Form	Frames	Date	Time User
1	80C8		0	c h p	1	0	1	29/07/93	08:56:59 dm
2	8548	3	0	c p	1	0	12	29/07/93	09:40:16 dm
3	8548		0	c p	1	0	1	29/07/93	09:40:57 dm
3 Queue Elements.					14 Frames in use.				

Figure 19.5: LISTPEQS

In order to find out what jobs are currently being processed by the spooler, any user may issue the:

LISTPEQS

command. A typical LISTPEQS report is shown in Figure 19.5. Here we see details of each report currently being processed by the spooler. Most of it is self-explanatory. Of interest are the LNK field which shows the number of the job which will be printed next, and the STATUS which will be a combination of letters including:

- c closed
- H hold file; this job is (or was) a hold file
- I immediate print (not being spooled)
- L locked

- N not closed
- O currently being output to the printer
- P waiting for printer
- R resubmitted for output
- S spooled
- T output to tape
- X killed.

indicating the current status of that job.

19.7 SP-EDIT

Any report or reports held on the form-queues can be handled by means of the:

`SP-EDIT {options}`

command. This allows you to inspect a held report, release it for output, transfer it to a normal Advanced Pick file, and/or delete it. A large number of options are available. Most frequently used are:

- n** to edit report number *n*. The report numbers can be read off the LISTPEQS display in Figure 19.5.
- n-m** to edit reports *n* to *m*.
- 'u'** to process all reports for user-id *u*. Note that the user-id is enclosed in apostrophes. Unless this or the U options are specified, only reports from the current user will be made available.
- D** to delete the report(s) without further intervention.
- F_n** to edit all reports on form-queue *n*.
- F_{n-m}** to edit all reports on form-queues *n* to *m*.
- L** to look at – but not manipulate – a report which is being output.
- M** is used with the S or D options to process multiple reports and suppress the input prompts.
- R** to reset the output form-queue number and the number of copies for the report to those of the current spooler assignments. Without this option, the report will be output with the original spooler assignments.
- S** to print the report(s) without further intervention.
- T** to direct the output to diskette/tape. This assumes a suitable diskette/tape has been loaded and attached. The **SP-TAPEOUT** command can later be used to retrieve the

report from diskette/tape and transfer it back to the spooler.

- U to edit all the available held reports. Unless the U or 'u' options are specified, only reports from the current user will be made available.

Some examples:

SP-EDIT

edit all held reports sent to the spooler by your user-id.

SP-EDIT 5

edit report number 5.

SP-EDIT 3-7

edit reports 3 to 7 inclusive.

SP-EDIT 5 T

send report number 5 to diskette/tape.

SP-EDIT 9 D

delete report number 9.

SP-EDIT 'DM'

edit all held reports sent by user-id DM.

SP-EDIT MS

despool all your held reports, sending them to the printer.

SP-EDIT DM

SP-EDIT MD

delete all your held reports, using the options D and M.

SP-EDIT MUD

delete all reports currently held by the spooler. The U option applies the deletion to all users' reports.

SP-EDIT MUS

print all reports currently held by the spooler.

When an SP-EDIT command is issued, for each report presented, the spooler will give you the opportunity to:

- o DISPLAY a part of the report,
- o Scan through the report to start output at a specific STRING,
- o SPOOL (that is, output) the report, and/or
- o DELETE the report.

If more than one report is to be edited, they will be presented for

processing in turn, one after another.

Your processing requests are specified in response to a sequence of prompts from the SP-EDIT processor:

1	DISPLAY?

At this point, you may enter any one of:

- Y to display the first 500 bytes of the report. If you wish to view more of the report, you can do so at step (3).
- N to pass to step (2).
- S to pass to step (3).
- D to pass to step (4).
- X to abandon the action.

<Return> will skip to the next report.

2	STRING

This allows to you to start printing at a specific point in the report, as you may want to do if the printer had run out of paper and you need to reprint just the last part of the report.

At this point, you may enter either of:

<Return> pass to step (3), or

xxxxxxx search for the string xxxxxxx and start printing at that point.

Be sure to enter a suitably unique identifying string. If your report has page numbers, you may be able to enter a string such as:

Page 34

Remember that spaces are significant and the string should be entered exactly as it appears in the report.

3	SPOOL?

At this point, you may enter any one of:

- Y to output the report.
- N to pass to step (4).
- T to output the report on the terminal.

TN to output the report on the terminal, with no pause at the end of each page. The T and TN option allow you to view the report before it is output.

F to save the output as an item (or items) on a file.

<Return> is equivalent to N.

If you enter F to save the output on a regular file, you will be asked to enter the name of the file:

FILE NAME?

and the first item-id which is to be used:

INITIAL ITEM NAME?

You should enter the appropriate information and the report will be transferred in *RUNOFF format* to the specified file. For example, if you entered:

FILE NAME? REPORT94
INITIAL ITEM NAME? FRIDAY

the first page will be saved as item FRIDAY on the file REPORT94, the second page as item FRIDAY0001, the third page as item FRIDAY0002, and so on. The full report may be output when required by means of a TCL command such as:

RUNOFF REPORT94 FRIDAY

to output the entire report, or:

RUNOFF REPORT94 FRIDAY004

to output the report starting at a specific page. The file, REPORT94 in this instance, must exist, and any existing items with item-ids FRIDAY, FRIDAY0001 and so on, will be overwritten by the spooler output.

4	DELETE?

At this point, you may enter:

y to delete the report from the queue.

Any other response will pass to the remaining report(s) specified on the SP-EDIT command, if there are any.

19.8 AP/Unix and the spooler

The features described above apply to all Advanced Pick systems, but AP/Unix systems are able to interface more closely with the Unix and the Unix spooler. To exploit this feature, the AP application still directs its

output to an AP form-queue, but an ASSIGNFQ command is used to link this with a Unix file or a specific Unix printer. This command has the forms:

```
ASSIGNFQ q,NETWORK(cat >fname)
```

```
ASSIGNFQ q,NETWORK(lp)
```

where *q* is the form-queue, *fname* is the name of a Unix file which is to receive the output, and *lp* is the Unix lp command which defines the printer to be used.

In both cases, once this linkage has been set up, form-queue *q* can be used exactly like a regular Pick form-queue.

The Unix LP command has several options, including:

- d xxx to link to a specific printer named xxx.
- m to send an electronic mail message when the job has been printed.
- nx to print x copies of the report.

19.9 Summary

The spooler is a standard part of the R83 Pick and Advanced Pick software which handles all reports sent to the printer(s). The spooler collects output destined for a printer and holds it on a disk file queue until the job is complete; the job is then released when the appropriate printer becomes free, or it may be held indefinitely for you to inspect / release / delete the report.

20

Backing storage

In addition to the hard disk, your Advanced Pick system will have one or more **magnetic storage devices** such as:

- 3.5" floppy diskette,
- 5.25" floppy diskette,
- 0.5" magnetic tape,
- 0.25" streamer cartridge tape,
- DAT / 8mm tape,

The chief use of these devices is in making backup copies of the contents of the hard disk, using the file-save and account-save features. They are also used in the archiving of data – such as last year's invoices – which are no longer required on the live system, and to transfer data between one Pick system and another.

We use the general term **backing storage** to refer to these devices. Data may be written to and read from backing storage by Basic programs and by standard Advanced Pick software, and the devices may also be controlled directly by TCL commands.

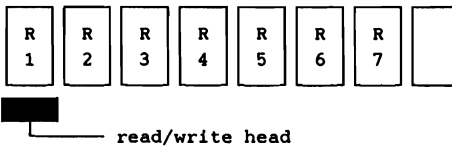
When you are using these devices on Advanced Pick releases before 6.1, it is important to remember that there is only a **single-user interface** for handling backing storage – this means that only one user may handle backing storage at any one time, and that user can use only one device at a time. On later releases, the **multiuser interface** allows a user to attach two or more devices, and allows two or more users to handle the various devices simultaneously.

20.1 Concepts

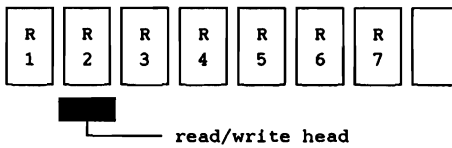
Advanced Pick regards each **volume** – be it a floppy diskette, a cartridge tape or a reel of magnetic tape – as a sequence of records, as illustrated in Figure 20.1. For a diskette, these records are 500 bytes in size; for a magnetic tape, their size is set by the T-ATT command.

When the volume is rewound – by means of the TCL T-REW command or the Basic REWIND statement – the volume is positioned so that the first record is at the read/write head, as in Figure 20.1 (a). As each record is read (or written), the volume is positioned at each record in turn, as

illustrated in diagram Figure 20.1 (b) where we see the position of the read/write head after the first record has been read. Figure 20.1 (b) reminds us that – if we are using a floppy diskette and we now remove the diskette and insert a new one – the reading/writing of the new diskette will commence at the second record. It is essential that a T-REW command be issued whenever a new diskette is loaded. Whilst the concept of rewinding a tape is easy to understand, the fact that a diskette must also be rewound is not intuitive and forgetting to do so can lead to mistakes and missing data.



(a) Backing storage after a T-REW



(b) Backing storage after a read/write

Figure 20.1: Records on backing storage

Whenever a file is written to backing storage, Advanced Pick creates an identifying record – the **header label** – in front of each file. Behind the header label come the individual data records, and every file is normally followed by an **end-of-file marker** – or **EOF**. Most Advanced Pick processes write an EOF marker after each file, and Basic programs use the WEOF statement for the same purpose. Labels and EOF markers are illustrated in Figure 20.2; physically, these are just like any other record, but their contents enable Advanced Pick to recognise them as labels and EOFs.

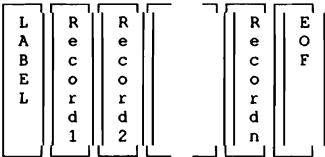


Figure 20.2: Labels & EOF markers

It is important to remember that Advanced Pick writes records to both floppy diskette and magnetic tape in this same manner.

20.2 Choosing the backing storage device

There are great many TCL commands for handling backing storage devices, and the same commands are used no matter whether you are using floppy diskette or magnetic tape. All the commands are of the form T-xxx and SET-xxx.

If your system has more than one backing storage device, you will first use one of the SET- commands to indicate which device you wish to use:

SET-8MM

sets the 8mm tape drive as the current backing storage device.

SET-DEVICE

specifies the current backing storage device which is to be used, as we see below.

SET-FLOPPY

sets the floppy diskette drive as the current backing storage device, as we see below.

SET-HALF

sets the 0.5" tape drive as the current backing storage device.

SET-SCT

sets the streamer cartridge drive as the current backing storage device, as we see below.

SET-SCT-DMA

sets the DMA (Direct Memory Access) channel for the streaming cartridge tape unit.

SET-TAPE-TYPE

specifies the type of half-inch tape drive which is being used.

EPick gives more details of these. Let's look at some of these more closely.

20.2.1 SET-FLOPPY

To use a floppy diskette, you will issue the:

```
SET-FLOPPY ({d}{r})
```

command, requesting that the diskette drive *d* be attached with recording density *r*. The drive letter will be A or B, and the density will be H for high or S for standard. For example:

```
SET-FLOPPY (HB
```

indicates that you are going to use drive B and that the recording density is to be high. If you only have one diskette drive, then forms such as:

```
SET-FLOPPY (H
```

are acceptable. Other options are described in EPick.

20.2.2 SET-SCT

To use a streamer cartridge tape, you will issue the:

```
SET-SCT {buffers} {(block-size)
```

command, requesting that the drive be attached and indicating the number of *buffers* and the *block-size* which are to be used.

20.2.3 SET-DEVICE

Keywords	Options	
720K	3	3.5" diskette
1.2M	5	5.25" diskette
1.44M	A	5.25" diskette
3 1/2"	B	3.5" diskette
5 1/4"	E	2.88MB diskette
FLOPPY	H	1.2MB 5.25" diskette
SCT	H	1.44 MB 3.5" diskette
8MM	L	720K 3.5" diskette
HALF INCH	M	720K 5.25" diskette
	S	360KB 5.25" diskette
	9	9-track 0.5" tape
	H	150MB 0.25" tape
	H	6250 BPI 9-track tape
	L	1600 BPI 9-track tape
	M	3600 BPI 9-track tape
	Q	0.25" tape
	S	60M / 120M 0.25" tape

Figure 20.3: SET-DEVICE keywords and options

AP/Unix systems use the SET-DEVICE command to attach the backing storage device. This combines the facilities of the individual SET-FLOPPY / SET-SCT / SET-HALF and other commands. The command has two possible forms:

```
SET-DEVICE {device} {(options)
SET-DEVICE {keyword} {(options)
```

where *device* is a device-number (0 to 15), and *keyword* and *options* are

any meaningful combination of those shown in Figure 20.3 together with:

n to select device number *n*.

null to display a list the available devices.

? to display the new settings.

c to change the block size on 8mm tape device.

F to specify the default floppy diskette.

I to suppress the display of devices.

K to specify hard disk pseudo-tapes (removable / fixed hard disks or regular Unix files).

N to specify network pseudo-tape in which it is assumed that another system is reading the data.

R to prevent an automatic rewind on the device. This is used with an unformatted floppy diskette.

The device-numbers and keywords can be displayed by means of the LIST-DEVICE command.

20.2.4 T-ATT / T-DET

Having indicated which device you are going to use, you may then need to attach the device to your processing and/or specify the block size; I say *may* because some forms of the SET command attach the device for you. This is done by means of the:

T-ATT

verb. If another user already has the device attached to *their* processing, then the system will respond with a message such as:

TAPE ATTACHED TO LINE 1

when you issue a SET- or T-ATT command and you will not be able to use the device. The U option may be specified if you wish to *steal*, the device from another user by using the U option:

T-ATT (U

When you have finished using the device, you should detach it from your processing by means of the command:

T-DET

thereby making it available for other users.

20.3 TCL commands for handling backing storage

The full range of commands used with the diskette/tape devices includes:

FORMAT {drive density {options}}

format the current floppy diskette. This command must be used to format a new diskette before it can be used by Advanced Pick. There is no equivalent command for magnetic tapes as these are not formatted.

T-ACT {device.number}

make a particular device the active device. On AP 6.1 and later, it is possible for a user to select and handle more than one backing storage device at one time. This command implies that all subsequent T- commands will manipulate this active device.

T-ATT {block.size}

attach the diskette/tape to the current port, thereby excluding other users, as described above. An optional block-size (in bytes) may be specified when attaching a tape device.

T-BCK {r}

T-BSR {r}

move the diskette/tape back a specific number of records. Here, as in other commands shown below, T-BCK is offered for compatibility with earlier releases, whilst T-BSR is a more memorable form consistent with the T-BSF command.

T-BSF {files}

move the diskette/tape backwards a specified number of files.

T-CHK

read a file on diskette/tape to check for and report any parity errors.

T-DET

T-DET {device.number}

detach the current diskette/tape and make the backing storage device available to other users. On AP 6.1 and later, it is possible for a user to handle more than one backing storage device at one time; in this case, the T-DET specifies the number of the device to be detached, if this is omitted, all devices currently attached to the user will be detached.

T-EOD

move the current volume to the end-of-data marker.

T-ERASE

erase the data on a tape. This is achieved by writing a empty block which is recognised as the end of data.

T-FSF {f}

move forwards a specified number of files.

T-FSR {r}

T-FWD {r}

move the diskette/tape forwards a specified number of records.

T-LINK dev1, dev2, ..., devn

link several backing storage devices together so that, when EOF is detected on one device, the system will proceed to the next. When the final device is reached, the system will stay with that device. If a looped link is specified:

T-LINK dev1, dev2, ..., dev1

the system will continue to use all devices in turn.

T-RDLBL

read the label on the current diskette/tape.

T-READ {n-m}

read and displays records from the current diskette/tape.

T-RET

T-RETN

retension a cartridge tape by winding and rewinding several times.

T-REW

rewind the current diskette/tape, positioning it at the load point.

T-SELECT

attach a diskette/tape. This is the fundamental command used by the SET-FLOPPY, SET-SCT and SET-HALF commands.

T-SPACE {f}

move the diskette/tape forward, skipping over a specific number of files.

T-STAT

T-STATUS

display the current status of all backing storage devices.

T-UNLD

T-UNLOAD

rewind the diskette/tape and then unload the volume, making it inaccessible for further processing.

T-VERIFY

read and verify the contents of a file-save or account-save volume.

T-WEOF

write an end-of-file marker on the diskette/tape. The standard

processes write EOFs with their files; the Basic WEOF statement can be used to write an EOF from a program.

T-WTLBL {xxxx}

write a label with optional text on the diskette/tape. The standard processes write labels with their files.

```
:SET-FLOPPY (AH
BLOCK size 500

:T-REW
BLOCK size 500
[96] BOT|
:T-DUMP STOCK WITH PAINT "RED"

    1 4500
    2 1234
    3 5555
    4 8000
    5 8765
5 ITEMS DUMPED.

:T-REW
BLOCK size 500
[96] BOT

:T-DET
```

Figure 20.4: Using a floppy disk

20.4 Using backing storage

A typical sequence of commands using a diskette is illustrated in Figure 20.4. The contents of the diskette produced here can be seen in Figure 20.5 where we discuss the Access T-DUMP and T-LOAD verbs.

20.5 Access and backing storage devices

There are several facilities which allow Access to handle data held on backing storage devices.

20.5.1 T-DUMP

The T-DUMP verb will allow you to copy all or selected items from a regular AP file to the backing storage device. This is often used to archive the contents of a single file – or even a single item – and is an alternative to the file-save and account-save activities. Since this is an Access verb, it may have forms such as:

```
T-DUMP STOCK
T-DUMP STOCK WITH TYPE "DESK"
T-DUMP STOCK WITH AGE > "300"
T-DUMP STOCK '1000' '2000' '3000'
T-DUMP STOCK 1000 2000 3000
T-DUMP STOCK > '1000'
T-DUMP STOCK BY DESCRIPTION
```

and so on. The equivalent S-DUMP verb will *sort* the items into sequence as they are dumped:

```
S-DUMP STOCK
S-DUMP STOCK WITH DATE < "1/1/95" BY DATE BY PRICE
S-DUMP INVOICES = "[95"
```

20.5.2 T-LOAD

The T-LOAD verb can be used to read a T-DUMP file and load the contents back on to a file on the hard disk:

```
T-LOAD STOCK
T-LOAD STOCK (O
```

to load the entire file, or to find and load only selected items:

```
T-LOAD STOCK '1000' '9000'
T-LOAD STOCK WITH PRICE > "50"
T-LOAD STOCK > '1000' WITH PRICE > "50" (O
```

The O option will cause any existing items on the disk file to be overwritten

STOCK DESCRIPTION..... LOCATION..		
1234	CHAIR, RED, OAK	DN/5/88
4500	CHAIR, RED, LAMINATE	LN/3/29
5555	TABLE, RED, PINE	LS/19/64
8000	SETTEE, RED, MAPLE	DN/17/39
8765	SIDEBBOARD, RED, MAPLE	DN/39/23
[405] 5 items listed out of 5 items.		

Figure 20.5: A T-DUMP file

20.5.3 TAPE modifier

Data which has been T-DUMPed to diskette/tape can be interrogated directly by Access sentences which include the TAPE modifier. For example:

SORT STOCK DESCRIPTION LOCATION TAPE

will produce an Access report using the DICT of the STOCK file to find the datanames DESCRIPTION and VALUE, but using the data on the diskette/tape, as illustrated in Figure 20.5. This assumes that the diskette/tape is positioned at the start of the appropriate T-DUMP file.

20.6 Summary

Advanced Pick will support floppy diskette and magnetic tape devices; these are known as backing storage devices. Only one user may use the backing storage at any one time, and only one device may be used at any one time. The data on these devices is regarded as a sequence of fixed-length magnetic records, and a range of TCL commands, Access verbs and Basic statements are available to manipulate / read / write the data on the devices.

21

Operations : Saving & restoring data

The nature of Advanced Pick does not demand a large operations department with responsibilities for running jobs, mounting tapes and diskettes, and so on; in a great many instances, the individual users are themselves the **operators**. This is true for the small single-user PC systems and also for many of the larger machines. Nevertheless, there is a need for *someone* to perform a number of domestic and control duties.

In addition to the domestic activities such as keeping the hardware clean and in working order, the tasks to be performed regularly as part of a standard rota include:

- Performing the file-save and account-save routines.
- Performing the file-restore routines.
- Monitoring the size and shape of the files on the various accounts, using the file statistics, reporting any notable situations and recommending that specific files be reorganised.
- Liaising with the maintenance engineer, arranging the periodic preventive maintenance visits and addressing any problems such as group format errors and others logged on the ERRORS file.

Additional activities to be performed less frequently as required or as necessary:

- Creating / deleting accounts.
- Creating / deleting user definitions.
- Archiving individual files.
- Restoring / recovering files and accounts.
- Trouble-shooting Activities: restoring the ABS frames, fixing group format errors, performing coldstart and warmstart.

The actual title of the person (or persons) responsible for these tasks depends upon the organisation. In this book, I use the term **System Manager**.

Here and in the following Chapter, we shall look at the more important duties of the System Manager. Let's start by looking at the file-save and file-restore operations.

21.1 Saving data

Probably the most important duty of the System Manager is that of making backup copies of the entire system and of individual accounts, and saving these copies on magnetic tape or floppy diskette. From time to time, it will also be necessary to recover all or a part of the system from these backup copies.

There are several ways of saving data on backing storage and of recovering data from the saved copies. These are shown in Figure 21.1.

	To save	To recover
The system	File-save	File-restore from file-save
An account	Account-save	Account-restore from account-save or file-save
A file	T-DUMP file	T-LOAD file from T-DUMP Selective-restore from account-save or file-save
An item or items	T-DUMP file item T-DUMP file i1 i2	T-LOAD file from T-DUMP Selective-restore from account-save or file-save
Changed items	FILE-SAVE with the option for an incremental save	Selective-restore from incremental save

Figure 21.1: Saving and restoring data

We discussed the S-DUMP / T-DUMP / T-LOAD verbs when we looked at the use of Access with backing storage devices in Chapter 20, and we shall look at the other utilities here.

21.1.1 File-save

The purpose of the file-save process is to save a copy of the contents of the entire system on backing storage. Such a **backup** of the system should be carried out at least once a week, but if it is feasible, it is wise to perform a file-save at the end of each working day. Systems which have significant activity – where a large amount of data is changed and processed each day – might be advised to perform file-saves more frequently.

The file-save process starts by looking at the MDS file at the head of the file system, as we saw in Chapter 4, and traverses the tree structure looking at each file on the system, saving the details of each file – such as the modulo and the lock-codes – and the contents of each file on backing storage as it goes. We shall look at this sequence more closely

in a moment.

21.1.2 FILE-SAVE

A file-save is performed by first selecting and mounting the appropriate diskette or magnetic tape, and then issuing the command:

FILE-SAVE

The process asks a number of questions:

- 1) Is this an incremental save?

We'll say more about **incremental file-saves** in a moment.

- 2) Console listing to printer?

If you want to **print** a list of the accounts and files as they are saved, enter Y.

- 3) Send statistics report to printer?

The file-save process produces a set of **file statistics** – including the size and number of items on the files – and saves these on the FILE-OF-FILES file. If you want to print a copy of these statistics at the end of the process, enter Y.

- 4) Bypass groups with GFE's?

If you want to be able to intervene and handle any **group format errors** which may be encountered, enter Y.

- 5) Do you want to sleep?

Take the default response of N, if you want the file-save to start immediately. If you want the process to wait until a specific time before starting, then enter Y here and, when asked, enter the time when the file-save is to start (the default is 23:00).

The file-save will then start as instructed. If the file-save requires more than one diskette or more than one tape, then you must change the volumes as and when instructed.

A file-save can be invoked via the standard menus ADMIN or ADMIN.FILES, or by using the SAVE verb and the appropriate options.

21.1.3 Account-save

An account-save performs a similar task to the file-save, but it does this for a **single account**, starting with the MD for the account and traverses the tree, looking at each file on the account, saving the size of each file and its contents on backing storage as it goes.

An account-save is performed by first selecting and mounting the appropriate diskette or magnetic tape, and then issuing the command:

```
ACCOUNT-SAVE
```

The process asks a number of questions:

1) **File-save tape label**

Enter any text which will be used to create a magnetic label to identify the diskette or tape later.

2) **Account name**

Enter the name of the account which is to be saved.

The account-save process will then proceed. If the account-save requires more than one diskette or more than one tape, then you must change the volumes as and when instructed.

An account-save can be invoked via the standard menus ADMIN or ADMIN.FILES, or by using the SAVE verb and the appropriate options.

21.1.4 Multiple account-save

The file-save and the account-save will process a select-list of accounts which are to be saved. Thus, a sequence such as:

```
SELECT MDS 'WAGES' 'PAYROLL' 'PAY-STATS'
FILE-SAVE
```

will save just the accounts WAGES, PAYROLL, PAY-STATS together with the DM account and the MDS file, and:

```
SELECT MDS 'WAGES' 'PAYROLL' 'PAY-STATS'
ACCOUNT-SAVE
```

will save just the accounts WAGES, PAYROLL, PAY-STATS. Such an operation is known as a **multiple account-save**.

21.1.5 The save sequence

If we consider the file structure shown in Figure 21.2 during a normal file-save, the file-save process will display a report like that in Figure 21.3, and the file-save tape will contain these items in the following sequence:

- 1) The file definition for the MDS file
- 2) The account definition for F1A
- 3) The file definition for F2A
- 4) The file definition for F3A
- 5) All the non-D-pointer items on F3A
- 6) All the non-D-pointer items on F2A
- 7) The file definition for F2B
- 8) All the non-D-pointer items on F2B
- 9) The data-level identifier for F2C
- 10) The data-level identifier for F3B
- 11) All the non-D-pointer items on F3B
- 12) The data-level identifier for F3C
- 13) All the non-D-pointer items on F3C

Any other files on account F1A will appear at this point.

- 14) All the non-D-pointer items on F2C
- 15) All the non-D-pointer items on the MD of F1A

The same sequence for all other accounts on the system.

- 16) All the non-D-pointer items on the MDS file.

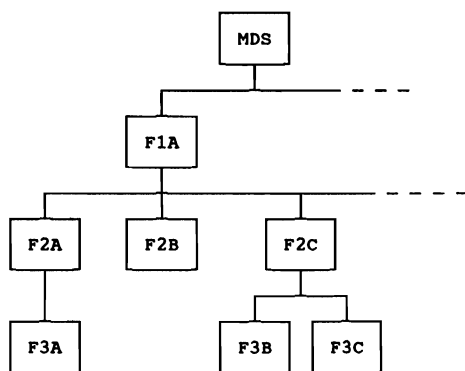


Figure 21.2: The file structure

```

1      2 MDS
1    339 MDS > F1A  7
1    340 MDS > F1A > F2A
  
```

| F1A account-save

```

1  343 MDS > F1A > F2A > F3A
1  341 MDS > F1A > F2B
1  342 MDS > F1A > F2C
1  344 MDS > F1A > F2C > F3B
1  345 MDS > F1A > F2C > F3C
1   37 MDS > DM
1   38 MDS > DM > NEWAC
1   39 MDS > DM > NEWAC > AT-DOS
1   40 MDS > DM > NEWAC > AT-NATIVE
1   41 MDS > DM > NEWAC > DELETE

```

Figure 21.3: Fragment of a file-save display

An **account-save** tape would contain just items (2) to (15) for the account. During a file-save, the display might be like that in Figure 21.3. A typical line of the file-save / account-save display:

```
1  344 MDS > F1A > F2C > F3B
```

shows the **reel number** (1 is the sequence number of the first diskette/tape), the **file number** (344 is the item-id of the item which identifies this file on the FILE-OF-FILES file), and the **path** for that file. The **file sequence number** which will be needed when performing a SEL-RESTORE with the N option is not shown on this report..

When going to a lower level, the D-pointers are chosen in their physical sequence; thus, if the D-pointer for F3C was physically before that for F3B on the DICT F2C, then F3C would be dumped before F3B.

21.1.6 DX-pointers and DY-pointers

This is a good place to remind ourselves about the DX-pointers and DY-pointers which can be used to define a file. These control the way in which the file and/or its contents are saved (or not saved) during a file-save or an account-save operation:

- o If the D-pointer for a file section includes an X in attribute 1:

```

DX
DLX
DPX
DSX

```

and so on, then that file section will be ignored completely during a file-save or account-save. If the account is subsequently restored from that save diskette/tape, then the file will no longer exist.

- o If the D-pointer includes a Y in attribute 1, then the file section itself will be retained during a file-save or account-save, but the **contents** of the file section will be ignored. If the account is subsequently restored

from the save diskette/tape, then the file will be restored, but it will be empty.

Look at Figure 21.4. Any of the D-pointers at (a), (b) or (c) may be a simple D-pointer, a DX-pointer, or a DY-pointer:

- a) A DX-pointer on the MDS file will ignore the entire account and its files. You could use this if you wanted to ignore a particular account during a file-save. The DX here has no effect if you perform an account-save.

A DY-pointer on the MDS file will retain the MD of the account but it will be empty (except for any D-pointers). Any files on the account and their contents will be saved in the normal manner.

- b) A DX-pointer on the MD will ignore the entire file, the DICT section and its data section(s). You will use this if you wish to ignore a file during a file-save / account-save.

A DY-pointer on the MD will retain the DICT section of the file but it will be empty (except for any D-pointers); any data sections and their contents will be saved in the normal manner.

- c) A DX-pointer on the DICT section of the file will ignore that data section of the file.

A DY-pointer on the DICT section of the file will retain that data section but it will be empty.

There are a number of facilities associated with DX and DY pointers. As we see in Chapter 4, the SET-DPTR verb can be used to change a D-pointer to a DX-pointer, DX to DY and so on. The X and Y options on the SAVE command will override any DX and DY-pointers, and the CHECK-DX utility can be used to list any DX-pointers on the system, as illustrated in Figure 21.5.

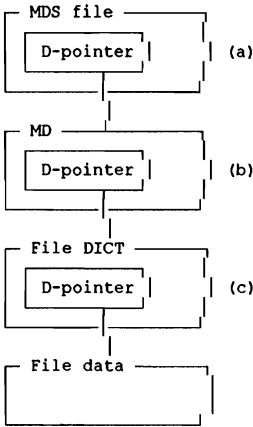


Figure 21.4: DX/DY pointers

```
DM
file-of-files
  file-of-files <----- file pointer is DX'ed
ts.log
  test <----- file pointer is DX'ed
  ts.log <----- file pointer is DX'ed
```

Figure 21.5: CHECK-DX

21.1.7 Incremental file-save

Every item holds a flag indicating whether it has been created, changed and/or deleted since the last file-save. The **incremental file-save** allows you to save only such flagged items and is performed by issuing the:

FILE-SAVE

command and giving the response Y when the process asks:

Is this an incremental save?

Because it is relatively quick, an incremental file-save can be performed more frequently than a full file-save. A typical sequence of events involving file-save and incremental file-saves is:

- o You perform a full file-save on Friday evening.
- o You perform an incremental file-save on Monday evening; this saves details of all changes which have been made since Friday evening.
- o You perform an incremental file-save on Tuesday evening; this saves details of all changes made since Friday evening (including the changes saved at Monday's incremental file-save).

and so on throughout the week. Then, if you need to recover lost data by means of a SEL-RESTORE selective restore, you can use the diskette or tape produced at Friday's full-save and/or at the most recent incremental file-save. A full file-restore and an account-restore can be achieved by restoring from Friday's file-save and then from the **latest** incremental file-save.

An incremental file-save is quite distinct from **transaction logging**, and the two processes are used independently.

An incremental file-save can be performed by selecting the FILE-SAVE option from the standard menus ADMIN or ADMIN.FILES, or by using the SAVE verb and the appropriate options.

21.2 File statistics

The FILE-OF-FILES holds information about each file on the system. We looked at this file in Chapter 4. The file is rebuilt when a file-restore is performed, and updated as new files are created, restored, cleared and/or deleted. The information held on FILE-OF-FILES is obviously important to the System Manager, indicating:

- o What files are on the system,
- o How big they are,

- Whether any files have unsuitable modules,

and much more. It is essential that the System Manager is familiar with this file and able to use it to produce reports showing the state of the file system.

The file statistics held on FILE-OF-FILES are generated as a by-product of the file-save process, and the printed report can be produced when the file-save is completed. The file statistics may also be produced independently at any time by commands such as:

```
SAVE (FS
```

to produce the file statistics for the entire system, or

```
SAVE (FSI
```

to produce the file statistics for an individual account.

In Chapter 4, we discussed the fields which are held on the FILE-OF-FILES and the datanames which can be used to produce reports on the file. The standard report can be produced by means of one of the commands:

```
LIST-FILE-STATS
LFS
LIST-FILE-STATS (P
LFS (P
```

and you can produce the file statistics for a single account by a sequence such as:

```
SELECT FILE-OF-FILES WITH MD "WAGES"
LIST-FILE-STATS
```

A typical LIST-FILE-STATS report is shown in Figure 21.6, and includes:

- The **reel** (or volume) **sequence number** of the diskette/tape where the file was saved.
- The **file sequence number** of the file on the diskette/tape. You will need this if you use the N option on the SEL-RESTORE command.
- The **file number** (on the FILE-OF-FILES file).

with subtotals for each account. Other specific reports can be produced by Access sentences using the available datanames. Note that the FILE-OF-FILES is not affected when the standard account-save or incremental file-save is performed; the S option on the SAVE verb will force the updating of the file statistics.

PAGE 1 ** FILE STATISTICS REPORT ** 29 Jul 1995									
R#	SEQ#.	FILE#	FNAME.....	MOD.....	BYTES.....	ITEMS..	OVFITMS	FRAMES..	PTRFMS
1	11	37	m/dict		29	86089	979	435	116
1	25	60	abs		1	1298	7		3
1	26	61	abs abs		397	1480169	6080	3281	1705
1	27	62	abs dbsym		307				307
1	62	86	bp		7	1153086	283	65	1159
1	63	87	bp bp		31	2324	6		32
1	65	89	bp dos		3	9589	7		13
1	66	90	bp includes		3	6164	3		10
1	64	88	bp unix.h		3	17705	9	4	24
1	26	47	cfg		1	36855	42	34	38
1	27	48	cfg cfg		1				1
1	29	50	cfg includes		1	2117	1		4
1	28	49	cfg pgm		1	10325	12	7	11

Figure 21.6: LIST-FILE-STATS report

R83 Pick users will recognise that the FILE-OF-FILES has many of the characteristics and applications of the STAT-FILE. The file has synonyms FF, FILES, FOF, and STAT-FILE.

21.3 The SAVE verb

The file-save / account-save / incremental file-save utilities which we have met in this Chapter are performed by macros which call up the fundamental SAVE verb. The SAVE verb performs a number of tasks and has a number of facilities which are not readily available via the standard utilities.

The general form of the SAVE command is:

SAVE (options

Some of the available options are shown in Figure 21.7. There are other options for logging and reporting group format errors, for saving a specified list of items on each file, and for clearing the dirty bits. EPick will give full details, and it would reward the System Manager to look more closely at the SAVE options which are available.

- B

save all B-tree indexes. This avoids having to regenerate the indexes after a restore.
- C

create a file-save / account-save which can be used to restore data back on to R83 Pick systems.
- F

display a list of the names of the files as they are processed. If this option is omitted, only the account names are displayed.

- I process a single account, as with the normal ACCOUNT-SAVE process. You will be asked for the name of the account to be saved. Without this option, a full file-save will be performed.
- Q display the account/file names in indented format as on R83 Pick systems, rather than the AP hierarchical format.
- S update the corresponding item on the FILE-OF-FILES for each file saved. This produces a new set of file statistics.
- T save the data on the diskette/tape. This option will be omitted if, for example, you simply want to produce the current file statistics by using the S option.
- U perform an incremental file-save.
- X save any files which are defined with DX-pointers, overriding the normal effect of DX-pointers.
- Y save any files which are defined with DY-pointers, overriding the normal effect of DY-pointers.

Figure 21.7: SAVE options

21.4 Verifying the save

It is important for the System Manager to be confident that the contents of a file-save / account-save / T-DUMP file are correct and that the data can be reread without problems. There are several ways of checking the integrity of a diskette/tape copy:

- o A file-save / account-save can be verified by performing a **dummy-restore** or by performing a T-VERIFY, and
- o A T-DUMP can be performed by performing a LIST or a T-CHK operation.

A **dummy-restore** is simply a selective-restore which reads through the entire file-save / account-save diskette/tape looking for a non-existent account. This can also be used to inspect the contents of the diskette/tape. A typical sequence for doing this is shown in Figure 21.8. The dummy-restore also shows the file numbers which you might need

if you wish to restore a single file by the SEL-RESTORE utility.

```
:T-REW
:SEL-RESTORE MD *
Account name on the tape: ZZZ          <== A non-existent account
File name: ZZZ                        <== Any file name
```

Figure 21.8: Verifying the save / dummy-restore

A T-VERIFY is more elaborate than a dummy-restore since it compares the data on the file-save / account-save file with the actual data on the system. This cannot be used successfully if there are other users on the system who may have changed the data since the save was carried out. EPick gives details of the various options available with T-VERIFY.

LIST (or indeed any Access sentence) with the TAPE modifier reads through a T-DUMP on diskette/tape reporting the items held there. This can also be used to inspect the contents of the file(s).

A T-CHK simply reads through any file on diskette/tape looking for the end-of-file marker.

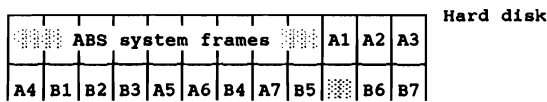
In all cases any physical errors – such as parity errors on the diskette/tape – will be reported. If there are any errors, the save operation should be repeated.

21.5 Restoring data

Having produced a file-save, an account-save or an incremental file-save, there may be a need to use these copies to restore data back to the system. The right-hand side of Figure 21.1 shows how we might use the available facilities.

21.5.1 File-restore

Having carried out a file-save and dumped a copy of the entire system and its files to diskette/tape, it is possible to recover the entire system from this copy. This process is known as a full-restore or a **file-restore**. As a file-restore takes place, the accounts and their files are restored in the sequence they are held on the file-save diskette/tape, and they are loaded sequentially on the system disk. When the file-restore is complete, the unused disk space will be available as a large block of contiguous frames.



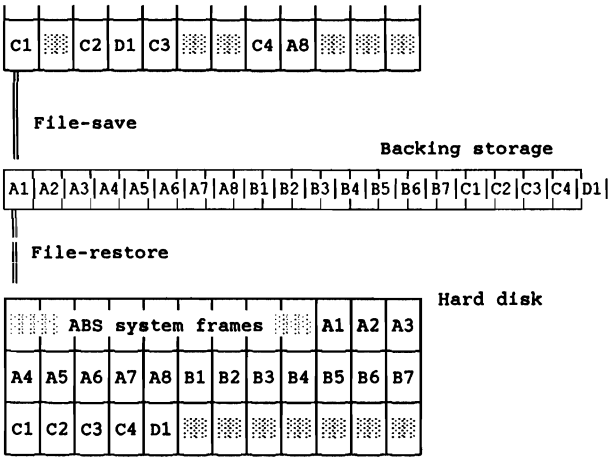


Figure 21.9: File-save / file-restore

This is illustrated in Figure 21.9 where we see how, before the file-save, the frames which belong to file A (numbered A1, A2, A3 and so on) are scattered about the disk, as are those for files B, C and D. There are also a number of unused (shaded) frames scattered about the disk; these are a consequence of some files and items having grown, and others having been deleted. When the file-save is performed, all the data for each file is dumped contiguously on backing storage, where we see all the frames for file A held together. When the file-restore is performed, each file is then restored to a contiguous set of frames on the hard disk. Note that the ABS system frames are not dumped to backing storage when a file-save is performed; the ABSDUMP verb is available for this purpose..

The :FILES command was available on R83 Pick systems to perform a file-restore, and is also available on later AP releases; on the first releases of AP, a file-restore was performed by taking the F option when the system is rebooted with a file-save diskette loaded.

Having completed the file-restore, the process will ask whether you want to restore further items from an incremental file-save and/or from a transaction log tape. When these have been done, the system is ready for use once more.

21.5.2 Account-restore

An **account-restore** is the process of recovering an entire account and its files from backing storage and reloading it back on to the system. You can restore an account from:

- o A file-save tape, or

- o An account-save tape, or
- o A mutiple account-save tape.

In most cases, accounts will be restored because they were deleted from the system previously; for example, you may have deleted the DEVELOPMENT account and now wish to restore it back to the system with the same name. It is, however, perfectly feasible to restore an account which already exists on the system; for example, you may have the WAGES account on the system and wish to restore an earlier account-save of the WAGES account, giving the new version the name ORIGWAGES, in order to compare the two or recover various pieces of data from the old copy.

The format of the ACCOUNT-RESTORE command is:

```
ACCOUNT-RESTORE account.name
```

and would be:

```
ACCOUNT-RESTORE DEVELOPMENT
and:
ACCOUNT-RESTORE ORIGWAGES
```

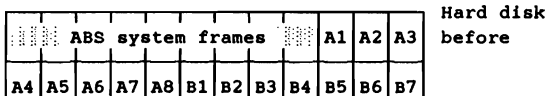
in the situations discussed above, specifying the name by which the account is to be restored to the system; as with ORIGWAGES, this need not be the same as the name of the original account held on the file-save / account-save tape, but it must not already exist on the system (that is, on the MDS file). The process will then ask for:

```
Account name on tape:
```

that is, the name by which the account can be identified on the file-save / account-save file. If you wish to recover the account with the same name as it was saved on backing storage, then you may just hit <Return> here. In our examples, the responses at this point would be DEVELOPMENT (or null) and WAGES, respectively. The account and all its files would then be restored back to disk, using up the available frames as required.

Having completed an account-restore, you will be asked whether you want to restore further updates from an incremental file-save and/or from a transaction log tape.

Figure 21.10 shows how the system might look after we had performed an account-restore from a file-save tape. In this illustration, we restore account X from account C on the file-save tape. We could not use the original name – C – because account C already exists on the system.



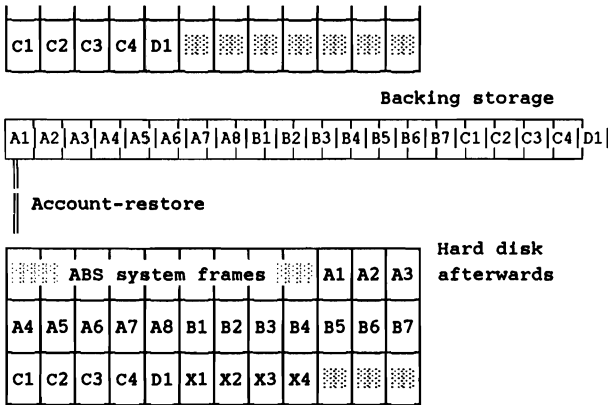


Figure 21.10: Account-restore

Account-restores can be invoked directly from TCL, or via the standard menus ADMIN or ADMIN.FILES.

21.5.3 RESTORE-ACCOUNTS

The RESTORE-ACCOUNTS utility reads through a diskette/tape containing a file-save or a multiple account-save, and restores to the system any accounts which are found on the diskette/tape but not on the system.

21.5.4 Selective restore

A **selective restore** allows an entire file or specific items of a file to be loaded back on to the system from a file-save or an account-save diskette/tape, and/or an incremental file-save tape, and/or a transaction log tape. The general form of the command is:

```
SEL-RESTORE {DICT} file.name {item.list} {(options)}
```

where *file.name* is the file on the system in which items will be placed (this file must exist on the system, although it need not necessarily be the same file name as that on which the items are held on the diskette/tape) and *item.list* is either the list of items to be restored, or null if you want to restore all the items saved for that file or if a select-list of item-ids is available. The available options are shown in Figure 21.11.

- o overwrites any items which already exist on the target file.
- A indicates that the diskette/tape is already positioned at the required

account.

- N specifies that the file on the diskette/tape is to be identified by its **sequence number** on the diskette/tape and not by its name. The sequence number is shown on the file statistics which are produced when the file-save / account-save is performed.
- C makes every item before the next end-of-file a candidate for restore. This ensures data can be restored even if a D-pointer is damaged on the diskette/tape.

Figure 21.11: SEL-RESTORE options

The SEL-RESTORE process first needs to know the format of the tape from which the data is being restored:

Restore from F)ull/account, I)ncremental, T)ransaction log

to which you should reply:

- F if the diskette/tape contains a full file-save or an account-save,
- I if it contains an incremental file-save, or
- T if it is a transaction log tape.

The SEL-RESTORE process will then ask:

Account name on tape?

to which you should reply with the name of the account with which the file was saved; if you specified option A or option N, the process will not ask for the account name.

The next prompt will be one of the following, depending upon whether the N option was used on the SEL-RESTORE command:

If the N option was used, the process will ask:

File#

and your reply will be the sequence number of the file on the diskette/tape. This is the SEQ# field on the file statistics report which was produced when the file-save / account-save was performed.

If the N option was not used, the process will ask:

File name?

This is the name of the file as it is to be found on the diskette/tape. Your reply will be one of the following:

null to restore the items from the MD of the account.

fffff to restore the items from the data section of the file fffff.

DICT fffff

to restore the items from the DICT section of the file fffff.

fffff sssss

to restore the items from data section sssss of file fffff.

The process will display the names of all the accounts and the files which it encounters as it searches the diskette/tape for the required file.

When that phase is complete and the item(s) have been restored, the process will ask whether you want to:

Restore from incremental save tape (y/n)?

and:

Restore from transaction log tape (y/n)?

allowing you to apply any later amendments to the items. If you had any such saves available, you would mount the appropriate diskette/tape and answer Y to these prompts.

Finally, a message of the form:

[992] n item(s) have been restored

or, if the end-of-file marker is found and the appropriate account or file or items have not been found, then the message will be:

[992] 0 item(s) have been restored

Selective restores are normally performed from TCL. Some sequences using SEL-RESTORE are shown in Figure 21.12.

- a) reads a file-save or account-save tape to restore all the items from the PAY-RATES file on the WAGES account and place these on the PAY-RATES file, overwriting any existing items.
- b) reads a file-save or account-save tape to restore all the items on the DICT of the STAFF file and place these on a file called WORKER. The diskette/tape is already positioned within the section for the correct account.
- c) reads a transaction log save tape to restore all the items from file number 334 and place these on the DICT of the INVOICES file, overwriting any existing items.
- d) reads a file-save or account-save tape to recover items 1000, 2000, 3000 and 4000 from the STOCK file on the SALES account and place these on the STOCK file.

(a)

SEL-RESTORE PAY-RATES (O

```

Restore from F)ull/account, I)ncremental,
      T)ransaction log (f/i/t): F
Account name on tape?  WAGES
File name?  PAY-RATES
Restore from incremental save tape (y/n)?  N
Restore from transaction log tape (y/n)?  N

```

(b)

```

SEL-RESTORE WORKER (A
Restore from F)ull/account, I)ncremental,
      T)ransaction log (f/i/t): F
File name?  DICT STAFF
Restore from incremental save tape (y/n)?  Y
Restore from transaction log tape (y/n)?  N

```

(c)

```

SEL-RESTORE DICT INVOICES (NO
Restore from F)ull/account, I)ncremental,
      T)ransaction log (f/i/t): I
File# 334
Restore from transaction log tape (y/n)?  N

```

(d)

```

SEL-RESTORE STOCK 1000 2000 3000 4000
Restore from F)ull/account, I)ncremental,
      T)ransaction log (f/i/t): F
Account name on tape?  SALES
File name?  STOCK
Restore from incremental save tape (y/n)?  N
Restore from transaction log tape (y/n)?  N

```

Figure 21.12: SEL-RESTORE

21.6 Transaction logging

Transaction logging is a feature whereby any changes made to a file are recorded immediately on backing storage. In the event of a system failure, the database may be restored from the last file-save tape and the logged transactions rolled forward to restore the system to its state at the time of the failure.

Transaction logging may be switched on/off and the scope of the logging can be all files or only those files which have DL as the first attribute of their D-pointer. The system will send the log records directly to backing storage or hold them on a disk file until the backing storage device is available.

Transaction logging may be invoked from the standard menu:

```
TXLOG
```

and there are a number of TCL verbs to support transaction logging:

```
LOG-STATUS
```

to display the status of the transaction logging system.

STARTLOG {{port}}{,wait}} {{options}}

to invoke the transaction logging system on the specified port (the default is the phantom port), allowing a specified number of milliseconds (the default is 30 seconds) for the tape buffer to flush.

STOPLOG

to terminate the transaction logging system and detach the backing storage device. Any subsequent log records will be written to a disk file. When the transaction logging system is restarted, these updates will be written to the transaction log tape.

TLOG-RESTORE

to restore all the transaction log records from backing storage. The log records may also be restored optionally after a file-restore / account-restore / selective restore has been performed.

TXLOG-STATUS

to display the status of the transaction logging system.

UPDATE-LOGGING

to enable / disable / report the status of the transaction logging system.

A file can be established with a DL-pointer by specifying the L option on the CREATE-FILE command which creates the file, or it may be assigned to an existing file by means of the SET-DPTR command, as discussed in Chapter 4.

21.7 File resizing

One of the System Manager's responsibilities is to ensure that all files are as efficient as possible in terms of the space they occupy and the speed with which items are read from the files. Both these factors are affected by the **modulo** of the file. The modulo of a file is established when the file is created and can only be changed when the file is restored to the system during a file-restore or an account-restore.

To indicate that a new modulo is to be assigned to a file when a restore takes place, you must set the **reallocation** field (attribute 13) of the D-pointer for the file to a value such as:

(123)

and then perform a file-save or an account-save, and at the next file-restore or account-restore, the file will be recreated with a modulo of 123. We discuss the choice of the modulo for a file in Chapter 5.

You can set the reallocation field manually:

UPDATE MD STAFF REALLOCATION

to change the size of the DICT section of the file STAFF; this is held in the file definition on the MD.

UPDATE DICT STAFF STAFF REALLOCATION

to change the size of the data section of the file STAFF; this is held in the data-level identifier on the DICT section of the file.

UPDATE DICT CARS DECEMBER REALLOCATION

to change the size of the data section DECEMBER; this is held in the data-level identifier on the DICT section of the file.

UPDATE MDS WAGES REALLOCATION

to change the size of the MD of the WAGES account; this is held in the account definition on the MDS file.

21.7.1 F-RESIZE

If you wish to automate this process, the:

F-RESIZE

command will determine a suitable modulo and reset the reallocation field for all files. The process asks:

List file changes to Terminal, Printer, no listing

and waits for your response T, P or null, and then reads through all the items on the FILE-OF-FILES, using the statistics about each file – the number of items and the size of the items – to determine a suitable modulo. This new modulo is then set in the reallocation field of the D-pointer for the appropriate file, and will be applied when that file is next saved at a file-save / account-save and restored to the system at a file-restore / account-restore. At the moment, there seems to be no way of restricting the scope of this resizing action.

The F-RESIZE process can be invoked directly from TCL, or via the standard menus ADMIN or ADMIN.FILES.

21.7.2 RESIZE

Later releases introduce a completely new feature which allows **immediate resizing** of a file: the RESIZE command. For example, let's assume that the STOCK file currently has a modulo of 11, then the command:

RESIZE STOCK (21

will effectively increase the modulo to 21 by appending a block of (in this instance) 10 frames to the file and amending the D-pointer to show the

original base FID and the FID of the new block. The before and after appearance of the D-pointer is shown in Figure 21.13. The two blocks of frames will be consolidated when the next save-restore is performed.

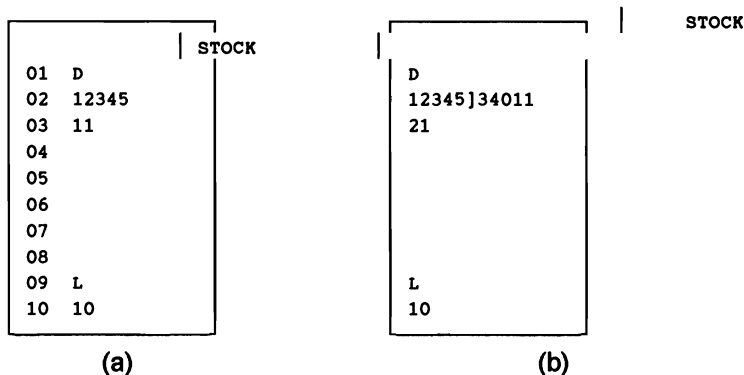


Figure 21.13: RESIZE and D-pointer

21.7.3 Resizing by hand

It is also possible to change the size of a file more quickly and directly by:

- o T-DUMPing the DICT section and the data section(s) to diskette/tape.
- o Deleting the original file.
- o Recreating the file with the required size for the DICT section and/or data section(s).
- o T-LOADing the contents of all sections of the file back from the diskette/tape to the newly-created file.

Of course, this should not be done if other users and their application programs are handling the file at the same time.

21.8 Summary

For the System Manager, the most important operational duty on an Advanced Pick system is to perform regular file-saves and/or account-saves, dumping a copy of the entire system (or of one or more accounts) to backing storage.

Less frequently, the System Manager will carry out a file-restore, rebuilding and reorganising the entire Advanced Pick database from the file-save copy.

The production of the file statistics, possibly using this information to change the modulo of existing files to make them more efficient in terms

of the disk space which they occupy and the speed with which their items can be read and written.

22

Operations : System administration

In the previous Chapter, we looked at the various ways in which the System Manager – and other users – can make backup copies of their data and recover data from these copies. In this Chapter, we look at some of the other – less frequent – duties which the System Manager might perform. These include:

- o Maintaining the users and the accounts on the system.
- o Monitoring the disk storage.
- o Verifying the accuracy of the standard system.
- o Monitoring and recovering from group format errors.
- o Monitoring the ERRORS file.
- o Setting the date and the time.
- o Creating and sending messages to users.
- o Producing and monitoring the accounting statistics.
- o Setting the users' terminal characteristics.

22.1 Starting and stopping the system

We shall not discuss the physical process of loading and installing the Advanced Pick system, nor shall we consider that of starting up the system; those tasks are described in the literature which accompanies your new AP system.

There are several points for the System Manager to note about the initialisation processes which can be performed during the day-to-day use of the system:

- 1) The SYSTEM-COLDSTART procedure on the MD of the DM account is invoked whenever the system is booted.

This is a good place to initialise the printers and carry out any other setup processes. Have a look at SYSTEM-COLDSTART on your system to see what it does. You may care to add to and/or to remove some of the processes which are included in the basis coldstart procedure; on my systems, I remove (by inserting COMMENT at the front of) the commands:

```
RUN DM,BP, SETPIO
RUN DM,BP, CLEANPIBS
VERIFY.SYSTEM
```

- 2) When a user logs on, the commands in the MACRO in his/her user definition on the USERS file are invoked.

This is a good place to put the TERM-TYPE command to set the terminal characteristics; a BULLETIN.BOARD command to pass any bulletin messages to the user. It is here that you might also include a LOGTO command to force the user to a particular account.

- 3) When a user logs to an account, the **logon procedure** for that account (this might be a macro, a menu, a Proc, or a catalogued program) will be invoked, if there is one. This logon procedure is an item on the MD which has the same name as the account (or the account synonym).

This is a good place to place any initialisation which is appropriate to the particular account. You might also include some secondary validation, possibly to request further identification or to ensure that the account is available to that user / that port / at that time of day.

We shall discuss many of these topics in this and the following Chapters.

22.2 Users and accounts

In order to use Advanced Pick, each user must provide two pieces of information:

- 1) His/her own **user-id**. Each user-id is defined by a user definition item held on the USERS file.
- 2) The **name of the account**, or master dictionary, which he/she wants to use. Each account is identified by an account definition item held on the MDS file.

When you install a new Advanced Pick system, a number of users are supplied:

DM
SYSPROG

and, in addition to the DM account, a number of other accounts may be supplied:

QA
PA
TUTOR

All other users and accounts must be created / maintained / deleted by the System Manager. Let's see how this is done.

22.2.1 Creating and changing users

Each individual user should be allocated his/her individual **user-id**. If users share user-ids, there may be conflict when they are attempting to use facilities, such as the TCL stacker, which use the user-id as a unique identifier.

User definitions are created by the System Manager – on the DM account – by means of the command:

```
UPDATE USERS user-id
```

The user-id may be any string of letters, numbers and other characters (but no spaces, and it's wise to avoid the " and ' and \ and * characters); typically, the user-id will be the user's name or initials. The Update Processor will then display the screen shown in Figure 22.1, and the standard <Ctrl> keys are used to maintain this item. Remember that you may enter ? at any point for an explanation of the meaning of that field.

name
address
zip
phone
keys
password
privilege
options
macro

Most of the fields are self-explanatory and the default values can be taken. Of special interest are:

Fig 22.1: USERS file

KEYS This field holds the user's **retrieval / update keys**.

These are used for **file security**: if a file has no **lock-code** or it has a lock-code which matches any of the user's keys, then access to the file will be granted for that user; if a file has a lock-code which does match with any of the user's keys, then access to the file will be prevented by that user. The keys may be null or there may be one or more multivalued keys.

PASSWORD This is a password which is required when the user logs on with this user-id. The password may be null. If there is to be a password, this will be added to the user item in its clear form (as it must be typed by the user) and will be stored and redisplayed in an encrypted form.

PRIVILEGE This is the **system privilege level** for this user and will be any one of SYS0, SYS1, or SYS2. The privilege level restricts the actions which the user can perform.

OPTIONS This may be null or any combination of the following:

- A Whenever this user logs on, the date, time and usage statistics will be logged on the ACCOUNTS file.
- M Any GFEs encountered by this user are to be automatically fixed and the offending frame(s) dumped to the ERRORS,GFE file.
- R This user's work is organised as a **closed system** and the user will

be thrown into the logon procedure if he/she types END from the debugger; this can be used to prevent the user from reaching TCL.

MACRO This is a sequence of TCL commands which are to be executed whenever that user logs on. When a new user is created, the macro includes the commands:

```
TERM-TYPE
SP-ASSIGN
BRK-LEVEL
ESC-DATA
SET-SYM GSYM
SPELLER-OFF
```

This list can be amended as required, and any additional commands added. Note that if a LOGTO (or TO) command is included at the end of this list:

```
LOGTO aaaaa
```

then that user will automatically be logged to account *aaaaa* after entering his/her user-id; this makes the logon process a one-step operation, much like the R83 Pick logon.

We meet some of these fields again when we discuss security in Chapter 23.

Any of the commands:

```
LIST USERS
LIST USERS (P
LISTUSERS
LISTUSERS (P
```

can be used to produce reports on the users currently defined on the USERS file.

User definitions can be amended / deleted by updating the USERS item:

```
UPDATE USERS user.id
```

and then deleted by means of the ^XO sequence.

You can also maintain users and the USERS file via the standard menus ADMIN or ADMIN.MAINT.

22.2.2 Creating and changing accounts

New accounts are created by the System Manager – on the DM account – by means of the command:

type
modulo
ret-lock
upd-lock
password
syspriv
justification
width
reallocation

Figure 22.2: MDS file


```
CREATE-ACCOUNT account.name
```

The *account.name* may be any string of letters, numbers and other characters. Typically, it will be a name such as PRODUCTION, SALES, ADMIN which describes the work done on that account. The update process will then display the screen shown in Figure 22.2. The standard <Ctrl> keys are used to maintain account definitions. Remember that you may enter ? at any point for an explanation of the meaning of that field.

The significance of the fields are:

TYPE This will be D if you are creating a true account or Q if you are creating an account synonym. The D may be followed by L if any changes to the MD of the account are to be recorded by the transaction logger.

MODULO This is the **modulo** of the MD of the new account. This field may be set when creating a new account, but it cannot be amended thereafter.

RET-LOCK

UPD-LOCK These fields hold the **retrieval / update lock-codes** protecting the account and its files, and are matched against the KEYS of the user's user-id whenever a file is handled.

PASSWORD This is a **password** which is required when any user logs on to this account. The password may be null. If there is to be a password, this will be added to the account definition in its clear form (as it must be typed by the user) and will be stored and redisplayed in an encrypted form.

SYSPRIV This is the **system privilege level** for the account and will be any one of SYS0, SYS1, or SYS2. The SYSPRIV level is superseded by the PRIVILEGE field in the USERS file.

JUSTIFICATION

WIDTH These specify the format in which the item-ids of the items on the MD are to be displayed. L and 12 are the defaults.

REALLOCATION This is a number in parentheses and specifies the modulo which is to be used for the MD when the account is next restored during a file-restore or account-restore. As we saw in Chapter 21, this will only be required when it becomes necessary for the System Manager to change the modulo of the account MD.

We shall meet some of these fields again when we discuss security in Chapter 23.

Any of the fields TYPE / RET-LOCK / UPD-LOCK / PASSWORD / SYSPRIV / JUSTIFICATION / WIDTH / REALLOCATION for an existing account may be changed by means of the command:

```
UPDATE MDS account.name
or:
CREATE-ACCOUNT account.name
```

In all cases, the familiar <Ctrl> keys are used to maintain the account definition.

Commands such as:

```
LISTFILES MDS
LISTFILES MDS (P
```

are used to produce reports about the accounts on the system.

Accounts can be deleted from the system by means of the:

```
DELETE-ACCOUNT account.name
```

command. The process will produce a list of the files owned by the account and then ask you to confirm that the account and all its files are to be deleted.

You can maintain accounts and the MDS file via the standard menus ADMIN or ADMIN.MAINT.

22.2.3 Account synonyms

It is possible to create **synonyms** for an account, allowing users to log to the same account by alternative names.

For example, we may have an account called WAGES, and this may have synonyms such as PAYROLL, TIME-SHEETS, CHIEF-ACCOUNTANT and so on. When a user logs to the CHIEF-ACCOUNTANT account, he/she will be passed to a **logon procedure** called CHIEF-ACCOUNTANT which may call up a certain program or present a certain menu, whereas a user logging to the TIME-SHEETS account will call up a different program or be presented with quite a different menu.

Account synonyms are held on the MDS file together with the account definitions, and may have any unique name. They may be created by means of the commands:

00	synonym name
01	Q
02	account name
03	
04	
05	
06	
07	
08	
09	L
10	12

(a)

type	Q
account	
justification	
width	

(b) U MDS synonym.name

Fig 22.3: Account synonyms

```
UPDATE ONLY MDS synonym.name
or:
EDIT MDS synonym.name
```

and have the format shown in Figure 22.3 (a). Alternatively, they may be created by means of the CREATE-ACCOUNT command, as follows:

- 1) Issue the command:

```
CREATE-ACCOUNT synonym.name
```

- 2) Enter Q in the TYPE field, and a PASSWORD, if you want one.
- 3) File the item, using the ^XF sequence
- 4) Add the account name to the synonym, by means of the command:

```
UPDATE MDS synonym.name
```

This will display the screen shown in Figure 22.3 (b).

- 5) File the completed synonym item, using the ^XF sequence.

The password on an account synonym may be set and/or changed by means of the PASSWORD command.

22.3 Details of the system

The WHAT command will display a number of system parameters and the current status of the system, as illustrated in Figure 22.4.

memory	pibs	pcb0	sysbase/mod	maxfid	available	dfsize	
1664k	74	2048	2394 7	28536	8123	1000	
System Locks							(a)
0	1	2	3	4	5	6	7 8 9 A B C D E F
0	###	###	###	###	###	###	###
1	###	###	###	###	###	###	###
Basic Locks							
0	1	2	3	4	5	6	7 8 9 A B C D E F
0	###	###	###	###	###	###	###
1	###	###	###	###	###	###	###
2	###	###	###	###	###	###	###
3	###	###	###	###	###	###	###
Spooler Locks							
mq	iq	fq	peq				

0 #####							
Group Locks							
PIB#		Lvl	Type	Filename			
Item Locks							
PIB#		Lvl	Hash	Item-id	Filename		
*000	0018DF	FF10	000018	T 1	ws.where1:000	ws.what ...	(b)
065	000841	BF10	000018	P	sp.sleep:027	sp.spool...	
073	0018E4	BF10	000018	P	pp.sched:02A		
The spooler is inactive.							(c)

Figure 22.4: WHAT report

The first part of this report shows:

- **MEMORY:** the size of RAM (the virtual memory) for the system.
- **PIBS:** the number of users (including the spooler and the phantom ports).
- **PCB0:** the FID of the start of the PCB (the primary control block) for port 0.
- **SYSBASE/MOD:** the base FID and modulo of the MDS file.
- **MAXFID:** the FID of the highest numbered frame on the system.
- **AVAILABLE:** the number of unused frames which are available for new files and for overflow. The OVERFLOW command gives a detailed breakdown of this space.
- **DFSIZ:** the size of the data frames. This report was produced on AP/Native where the frames are 1000 (or more precisely 1024) bytes in size.

Then follow three further reports:

- a) The **lock tables**, as output by the LIST-LOCKS command. This is section (a) of Figure 22.4 and shows any locks which are currently set.
- b) The **user status report**, as output by the WHERE command, shows the AP routines which each user is currently executing.
- c) The **spooler status report**, as output by the SP-STATUS command, shows the status of the printers and the spooler.

The general format of the command is:

```
WHAT {(options)}
```

and the options shown in Figure 22.5 allow you to tailor the report.

n{-m} to output the WHERE report for ports *n* to *m*.

'uu' to include the WHERE report only for users logged on to user-id *uu*.

L to suppress the output of the lock tables, section (a).

N to suppress the pause at the end-of-page when displaying on the screen.

P to send the output to the printer.

S to suppress the output of the SP-STATUS report, section (c).

W to suppress the output of the WHERE report, section (b).

Z to include the WHERE report for all lines, whether they are in use or not.

Figure 22.5: WHAT options

22.3.1 Details of disk storage

There are several commands for monitoring disk usage and the fragmentation of the available disk space. As Figure 22.4 illustrates, part of the WHAT report shows the total number of frames of disk storage which are unused.

The OVERFLOW command will give a more detailed picture of the unused frames, as illustrated in Figure 22.6. In (a), we see a summary of the number of frames available. In (b), we see the size and location of each individual block of unused frames, and in (c) we see just the size of the available blocks. If this report indicates that the available disk space has become too fragmented – that is, with a large number of small blocks – it may be that there is no single block of frames sufficiently large to enable you to create a new file. In such circumstances, it may be necessary to perform a file-save and file-restore to consolidate the disk space.

overflow: 8213 reserve: 1024 blocks: 34 b-tree levels: 1 leaves: 1

(a) OVERFLOW

6393- 6395 : 3	6397- 6398 : 2	6401- 6405 : 5
6407- 6412 : 6	6414- 6414 : 1	6431- 6437 : 7
6439- 6439 : 1	6441- 7482 : 1042	8996- 8998 : 3
9555- 9560 : 6	9564- 9566 : 3	10610-10610 : 1
10618-10618 : 1	10620-10620 : 1	10634-10634 : 1
13857-13857 : 1	14664-14664 : 1	14714-14714 : 1
14716-14716 : 1	14718-14718 : 1	14720-14720 : 1
14724-14724 : 1	14728-14728 : 1	14745-14745 : 1
14753-14753 : 1	14769-14769 : 1	14772-14772 : 1
14775-14775 : 1	14833-14838 : 6	14844-14844 : 1
14874-15301 : 428	15303-15303 : 1	15305-15306 : 2
21858-28536 : 6679		
overflow: 8213 reserve: 1024 blocks: 34 b-tree levels: 1 leaves: 1		

(b) OVERFLOW (A

3	2	5	6	1	7	1	1042	3
6	3	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	6	1	428	1	2	6679		
overflow: 8213 reserve: 1024 blocks: 34 b-tree levels: 1 leaves: 1								

(c) OVERFLOW (B

Disk Available				
0....1....2....3....4....5....6....7....8....9....100%				

		MB	Bytes	Frames
Maximum disk space	100%	29	28,536,000	28,536
System usage	8%	2	2,394,000	2,394
Data usage	63%	18	18,019,000	18,019
Available disk space	28%	8	8,123,000	8,123

(d) FREE

Figure 22.6: OVERFLOW report

These particular OVERFLOW reports indicate that there are 1024 frames held in **reserve**. When a file-restore is performed, the reserve pool is set to this level. In an emergency, frames can be transferred from the reserve pool to the general pool of available overflow frames by means

of the SET-OVF-RESERVE command. For example, the command:

```
SET-OVF-RESERVE (500)
```

would reduce the reserve pool to 500, transferring 524 frames to the overflow pool. This is a temporary measure and a file-save and file-restore should be carried out as soon as possible. It is unwise to run the system with no reserve frames.

The FREE command gives a visual presentation of the available disk space, as illustrated in Figure 22.6 (d). This is similar to the DISK-USAGE command of Release 3.1 of R83 Pick.

22.3.2 Verify the system

Recall that the ABS frames hold the code for all the standard AP facilities. It is essential that the contents of the ABS frames are correct and, from time to time, the System Manager should verify them by means of the:

```
VERIFY-SYSTEM
```

or the equivalent VERIFY-ABS command. The process will compare the check-sum of each frame with the expected figures held in the ABS file, and report any errors. You may carry out this operation at any time – even whilst there are other users logged on – and you can abandon the verification process by hitting the X key.

If any errors are detected during the VERIFY-SYSTEM, the:

```
:ABSLOAD
```

command should be used to reload the ABS frames from a file-save tape or from the original Pick system diskettes/tapes. This should not be done whilst there are other users on the system.

22.3.3 Group format errors

A **group format error – GFE** – is a serious error in the data content of a frame of virtual memory. As a result of a GFE, the Pick system is unable to reach and process data which lies in the group of frames beyond the site of the error. We shall have more to say about GFEs in Chapter 24.

GFEs are reported as they are encountered during processing and the file statistics report shows any GFEs detected during a file-save or account-save.

If you suspect that a certain file may have GFEs – you may have been working on it when there was a power failure – then you can scan it by means of a command such as:

```
LIST STAFF
LIST ONLY STAFF
COUNT STAFF
```

which forces the Access processor to read all the items on the file. If you wish to check for any GFEs in the frames used by indirect items, then you might use a command such as:

```
LIST STAFF WITH A1
LIST ONLY STAFF WITH A1
COUNT STAFF WITH A1
```

You can scan all the files of a suspect account for GFEs by means of the:

```
CHECK-ACCOUNT
```

or the equivalent:

```
SAVE (FI
```

command; both perform a **dummy account-save**. You can check the entire system for GFEs by means of the commands:

```
CHECK-FILES
OF::
SAVE (F
```

either of which will perform a **dummy file-save**.

In all cases, if the process runs to completion without any errors being reported, then there are no GFEs on the file / account / system.

22.3.4 ERRORS

Whenever any system errors, such as group format errors, forward link errors, or backward link errors, are detected, Advanced Pick logs these on the ERRORS file. Any messages logged by the LOG-MSG command are also recorded on this file. The System Manager should monitor the ERRORS file on a regular basis, using the command:

```
LIST ERRORS
```

investigating any suspicious situations and reporting any mechanical errors or system errors to the maintenance engineer.

When any GFEs are reported and/or fixed by the GFE handler, a dump of the offending frames will be placed on the ERRORS,GFE data section

of the ERRORS file. This, too, should be monitored by the System Manager:

```
LIST ERRORS,GFE
```

22.4 The date and the time

Once the date and time of the Advanced Pick calendar/clock have been set, they should need little attention. The current setting for both date and time can be checked by means of the:

```
TIME
```

command.

The system date can be reset by means of the SET-DATE command. This has a form such as:

```
SET-DATE 1 JAN 1999  
SET-DATE 1/1/99
```

which will reset the system calendar until the next coldstart, at which point the date will be reset from the hardware calendar/clock. If the U option is specified:

```
SET-DATE 1 JAN 1999 (U  
SET-DATE 1/1/99 (U
```

then the hardware calendar/clock will also be updated.

If the form **1/1/99** is used, care should be taken to ensure that the correct date format – European or USA – is in force. The external form of the date can be set by the commands:

```
SET-DATE-EUR  
to set the European format: 31/12/99
```

```
SET-DATE-STD  
to set the American format: 12/31/99.
```

The system clock can be reset by means of the SET-TIME command. This has a form such as:

```
SET-TIME 09:30:59 (U  
SET-TIME 09:31 (U
```

As before, the U option will update the hardware calendar/clock. Otherwise, the system clock will only be reset until the next coldstart, at which point the time will be reset from the hardware clock.

22.5 Messages

A user may communicate with any other user by sending a message with the MESSAGE (or the equivalent SEND-MESSAGE and MSG) command. Some examples are:

MESSAGE * message

pass the *message* to all users who are currently logged on, including the line which issued the MESSAGE command.

MESSAGE JOE message

pass the *message* to all users logged on with the user-id JOE. The command returns a message if there are no such users logged on to the account.

MESSAGE 17 message

pass the *message* to port number 7, even if there is no user logged on there.

MESSAGE !* message

pass a *message* to all ports, whether or not there is a user logged on there.

The MESSAGE command should be used with caution as it will be output on the target terminal(s) with no ceremony, possibly upsetting the user's screen display. If a user receives a message as he/she is using the Update Processor or the Menu processor, the screen will be refreshed after the message has been accepted.

Figure 22.7 shows the places within the standard system where the System Manager may insert messages or modify the standard messages. These messages are held in the usual MESSAGES file format, as described in Chapter 4, and will be displayed to all users during the log on / log off process. The most important of these are:

- o LOGON message on the MDS file. This is the message which is displayed as each user is invited to log on to the system. Note that this is held in the usual message format but it is to be found on the MDS file with the item-id LOGON. This is a useful place for putting your organisation's logo or other identifier.
- o LOGON message on the MESSAGES file. This is the message which is displayed after the user has successfully entered his/her user-id (and password). If no message is required, this item should contain null text or just an L parameter.
- o Message 335. This is the message which is displayed after the MESSAGES LOGON and before control is passed to TCL (or the

logon procedure for the account). If no message is required, this item should contain null text or just an L parameter.

- Message 340 is displayed when the user logs to another account.
- Messages 340 and 336 are displayed when the user logs off.

The sequence in which these messages are presented is shown in Figure 22.7.

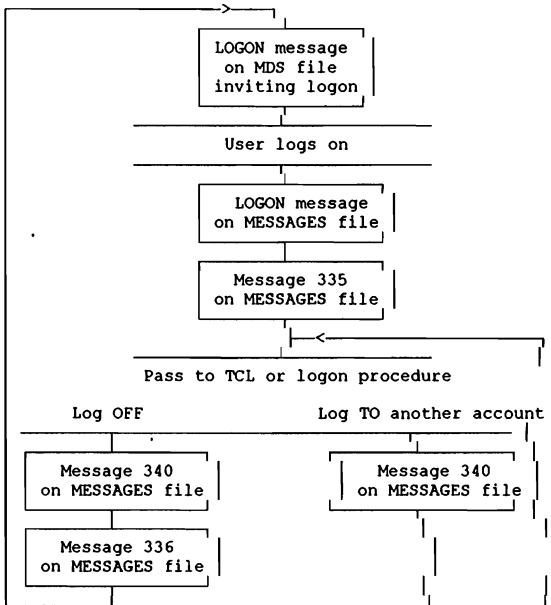


Figure 22.7: The messages

There is also the BULLETIN file which is used to hold messages which are to be displayed to users. The BULLETIN items are created / maintained by a command such as:

```
UPDATE BULLETIN xxxx
```

and contain four attributes: attribute 1 holds the user-id of the sender of the message; attribute 2 holds a brief title for the messages; attribute 3 holds the text of the message in Output Processor format; attribute 4 is used by the system. The bulletins are displayed by the command:

```
BULLETIN.BOARD
```

to display all available messages, or:

```
BULLETIN.BOARD NEW
```

to output any messages which have not already been displayed to this user. In this case, the user-id of the recipient of the message is placed in attribute 4 of the BULLETIN item.

The BULLETIN.BOARD command can be issued at any time and may be included in the macro in the users' definitions items or in the logon procedure.

22.6 Accounting statistics

As each user logs on and logs off the system, Advanced Pick monitors this by creating and updating certain items. Thus, if user *uuuu* logs on at port *pppp*, the following files are updated:

- 1) Item *pppp* on the PIBS file is updated to show the user-id, the account-name, the logon date and the logon time. This information is used by the LISTU command.

... and, if there is an A in the options field (attribute 9) of the USERS item for user *uuuu*, then ...

- 2) Item *uuuu* on the ACCOUNTS file is updated to show the account-name, the logon date and time. When the user logs off, this item is updated to show the connection time, the number of charge-units (processing time) used, and the number of pages of printed output produced. This information is used by the CHARGES command.

Some organisations use the accounting information produced at stage (2) as a basis for interdepartmental charging for computer services. A summary of the current accounting information can be produced by the LISTACC command:

```
LISTACC
LISTACC (P
```

Unless you actually need to use the accounting information, it is better not to produce it; recording the information needlessly takes up time and disk space. Accounting for an individual user may be suppressed by removing (or not entering) A in the OPTIONS field of the USERS item; accounting may be suppressed across the system by removing the ACCOUNTS file. If you do produce the accounting information, it is good practice to print the LISTACC report and clear the ACCOUNTS file down from time to time.

As a user logs off, the usage (that is, the total connect time, the CPU time and the number of printed pages) will be displayed:

```
<Connect Time=12 Mins; CPU=345 Units; LPTR Pages=6>
```

This same information can be displayed at any time during the session by means of the CHARGES command.

If you wish to break down your accounting information, possibly charging certain work to a particular client, you can achieve this by using the:

```
CHARGE-TO wwtools
```

command. This will effectively extend the user-id to:

```
uuuu*wwtools
```

amend the PIBS file (as in stage (1) above) and (provided that the accounting information for user-id *uuuu* is being logged) will create a new item on the ACCOUNTS file with the item-id *uuuu*wwtools* and all subsequent usage will be charged to that item. This extended user-id will be shown on the WHO command (and by the U50BB and U60BB user-exits) and should be used in the MESSAGE command, but the original user-id *uuuu* will be retained to identify the TCL stacker list of commands and to identify spooler jobs.

22.7 Terminal characteristics

In order to control the output to each terminal, Advanced Pick holds a number of pieces of information about each port on the system, as shown in Figure 22.8.

```
:TERM

Terminal name: MM-MON
Product name: IBM VIDEO
Terminal width: 79      printer width: 80
           depth: 25      depth: 58
Lineskip:      0
Lf delay:      7
Ff delay:      2
Back space:    8

:TERM-TYPE

Term setting: 79,25,0,7,2,8,80,58,MM-MON
```

Figure 22.8: Terminal characteristics

These parameters are known as the **terminal characteristics** and are held in the DEVICE field of an item which is held on the PIBS file for the each port, and the settings are imposed when a TERM-TYPE command is issued from that port. The terminal characteristics include:

- **Terminal name:** this is a code denoting the type of terminal which is being used on the port. This name – MM-MON in the example shown here – is the item-id of an item on the DEVICES file which holds details of the escape sequences and other codes which are used to produce effects, such as cursor positioning and clearing the screen, on that type of terminal. A set of DEVICES items are provided for the more frequently-encountered terminals (and printers), and the System Manager can add to these by means of the DEFINE-TERMINAL utility. The **product name** is taken from the MM-MON item on the DEVICES file.
- **Width:** the number of characters per line for the devices. There are two width parameters, one for the terminal and one for the spooler/printer.
- **Depth:** the number of lines per page (or screen) for the device. Again, there are two depth parameters.
- **Line skip:** the number of lines to be left at the foot of the screen. This is the difference between the page depth value for the screen and the physical depth of the screen. This parameter is usually set to 0.
- **Line-feed delay:** the number of delay characters which are to be output after each line-feed when sending output to the screen. This

parameter is usually set to 0.

- o **Form-feed delay:** the number of delay characters which are to be output after each form-feed, or top-of-form skip. This parameter is usually set to 2. A value less of 0 or 1 here will cause the screen output to scroll (instead of clearing the screen) between pages.
- o **Backspace:** this is the decimal value of the character which the terminal is to interpret as the backspace character. Normally, this will be 8.

The terminal characteristics are used to determine the format (width and depth) of all output reports and also the terminal display protocol. It is, therefore, essential that the characteristics are set correctly for each port on the system.

This is particularly true of the terminal name: if the characteristics shown here (for terminal type MM-MON, a memory mapped monitor terminal) were to be set at a port which was connected to, say, a Wyse 50 terminal, then much of the screen output would be unintelligible. The reason for this is that, whenever an instruction such as *clear the screen* is executed, Advanced Pick looks at your terminal characteristics to determine what sort of terminal you are using, then it looks to the DEVICES file to see what sequence of characters will clear the screen on that particular piece of equipment, and finally it sends the required string of characters to clear the screen. In this particular instance, the characters:

```
HEX(1B) HEX(2A) HEX(1)
```

will clear the screen on terminal type MM-MON, but the character:

```
HEX(1A)
```

clears the screen on a Wyse 50 terminal but has no special effect on a terminal type MM-MON.

There are several commands for changing the terminal characteristics: TERM and SET-TERM. TERM is used to set the characteristics for the current port and these remain in operation until another TERM or another TERM-TYPE command is executed. The TERM command has a form such as:

```
TERM 79,25,0,7,2,8,80,58,MM-MON
```

Compare this with the resultant settings shown in Figure 22.8. Any parameters which are not to be changed may be omitted (although the positional commas must be included). So we might issue:

```
TERM ,,,,,,132
```

if we wanted to change only the printer width.

SET-TERM has the same general form and is used to set the default

characteristics for *all* ports; these default characteristics will be assigned at logon and remain in effect until they are reset by a TERM-TYPE command.

When the system is initially set up, the System Manager will assign the characteristics for each port by means of commands of the form:

```
UPDATE PIBS DEVICE 'n'
```

where *n* is the appropriate port-number. The characteristics set in this manner will apply when a TERM-TYPE is next executed on port *n*. There should be a TERM-TYPE command in the logon procedure for each user and/or for each account. The DEVICE field of the PIBS file (especially the terminal name) should be changed whenever the physical device attached to a port is changed.

The SET-PORT and the SET-BAUD commands are used to set the communication characteristics and the baud rate for a port. Full details of these commands can be found in EPick.

22.8 Some further topics

In this Chapter, we have looked at a number of topics which are important to the System Manager. Other topics and commands which might be of interest later are:

- o Locking
- o The monitor
- o Virtual machines
- o TCL commands: BUF-MAP / BUFFERS / BUFFERS.G / CLEAR-LOCKS / CONFIG CORE / CONVERSE / DISC / FLUSH / INIT-OVF / MIRROR / MONITOR-STATUS / SET-BATCH / SET-BATCHDLY / SET-BAUD / SET-FLUSH / SET-PORT / SHPSTAT / TANDEM.

You can find details of these in EPick.

22.9 Summary

The system manager is responsible for a wide range of routine and administrative duties: creating and maintaining users and accounts; monitoring disk storage; verifying the accuracy of the standard system; monitoring and reacting to errors reported on the ERRORS file; setting the date and the time; creating and sending messages to users; producing and monitoring the accounting statistics; setting the users' terminal characteristics.

23

Security

What do we mean by security? For the purposes of this Chapter, we shall use the word **security** to mean the System Manager's ability to control the users' access to the system, and to control and monitor what users can and cannot do to the database. We shall look at:

- o Logging on and passwords.
- o System privileges: controlling what users can do.
- o Lock-codes: controlling the user's access to others' files.
- o Logon procedures and closed systems: constraining the users to their application system.
- o Controlling the MD: controlling what the users can and cannot do.
- o Monitoring what the users are doing and have been doing.

23.1 Logging on and passwords

There is a certain security in the fact that a **user-id** and an **account name** or an **account synonym** may be any string of characters and are virtually unrestricted in length and format.

Beyond this, each user-id, account name or account synonym may have a password. Passwords may be set and changed by means of the **PASSWORD** command. This asks:

```
Change USER or ACCOUNT password (U, <A>)?
```

and then:

```
Enter user name
```

OR:

```
Enter account name
```

Having indicated which user-id / account name you wish to handle you must then enter the new password (or null, if you are removing an existing password), and then enter it once again for confirmation. If you enter null for the new password, you will be asked to confirm that you wish to remove the password.

Advanced Pick keeps no record of who has attempted to access the system or of failed attempts to log on. It is possible, however, to implement this through a program invoked from the macro of the user definition or during a logon procedure, as mentioned below.

23.2 System privileges

System privileges restrict the actions which a particular user can perform, irrespective of the verbs which are to be found on the MD of the account, thereby allowing the System Manager to restrict the general level of activity for that user. The system privileges are set when the USERS item is created or changed:

```
UPDATE USERS 'uuu'
UPDATE USERS 'uuu' SYSPRIV
```

where *uuu* is the user-id, and will be any one of SYS0 or SYS1 or SYS2. The SYS*n* indicator is held in the SYSPRIV field of the items on the USERS file. The possibilities are:

- **SYS0** is the lowest level of security and SYS0 users:
 - Cannot use the backing storage devices.
 - Cannot update the MD or other dictionary files.
 - Cannot create, clear or delete files.
 - Cannot create, clear or delete indexes.
 - Cannot start/stop printers.
 - Can only use the P, G, END and OFF commands when in the debugger.
- **SYS1** users:
 - Can use backing storage devices.
 - Can use the verbs: CREATE-FILE, DELETE-FILE, CLEAR-FILE, CREATE-INDEX, CLEAR-INDEX and DELETE-INDEX,
 - Cannot perform save/restore operations.
 - Can update the MD and dictionary files.
 - Cannot amend the MDS file.
 - Cannot start/stop printers.
 - Can only use the P, G, END and OFF commands when in the debugger.
- **SYS2** users have access to all Advanced Pick facilities, including file-save and file-restore operations, use of the Assembler and any debugger commands. The following can only be performed by a user with SYS2 privileges:
 - DUMP
 - CLEAR-LOCKS
 - INIT-OVF
 - LOGOFF
 - SET-BAUD; SET-PORT; SET-TIME; SET-DATE
 - SP-KILL
 - STOPPTR; STARTPTR
 - CONVERSE; MIRROR; TANDEM.

Since there are only three such levels, the distinction is fairly coarse. If a user attempts to perform an action for which his/her privilege level is

inadequate, then the message:

YOUR SYSTEM PRIVILEGE LEVEL IS NOT SUFFICIENT FOR THIS STATEMENT

is displayed and the operation will be abandoned.

There may also be a SYSPRIV field on the account definition items on the MDS file, but this is ignored.

23.3 Lock-codes

In the simplest situation, the files which belong to one account are inaccessible to users of other accounts. For example, a user who is logged on to the PERSONNEL account cannot look at the contents of the INVOICES file which has been created on the SALES account.

However, this is not an ideal situation. If accounts were not able to share files, it might mean that there were several copies of the same information on the system, each copy being accessible by a certain account; if we did this, there would be the added problem of ensuring that the various copies were up-to-date and consistent. The solution is to create a pointer which behaves like a D-pointer and is used like a D-pointer, but which points not to a specific frame of virtual memory but holds the account name and the file name which you want to access. Such an item is called a **Q-pointer** or a **file synonym**, as we saw in Chapter 4.

File Q-pointers will allow users of one account look at, and use, files which belong to another account. But how can you restrict the access? How can you stop people amending files which they are only supposed to look at? The Advanced Pick answers to these questions are **lock-codes**.

By setting lock-codes on your files you can stop other users looking at your files or changing them. Only users who have matching **keys** in the KEYS field of their item on the USERS file will be able to handle any locked files. Figure 23.1 shows that lock-codes may be held on any – or all – of:

- The **account definition** on the MDS file: a lock-code here will lock the MD of the account, and hence the dictionaries and data sections of all files on that account.
- The **file definition** (the D-pointer) on the master dictionary which identifies the file: a lock-code here will lock the DICT section of the file, and hence all data sections which depend upon that DICT section.
- The **data-level identifier** (the D-pointer) on the dictionary of the file: a lock-code here will lock that data section of the file.

A lock-code in any of these positions will lock all linkages below that point, as the vertical bars in Figure 23.1 indicate.

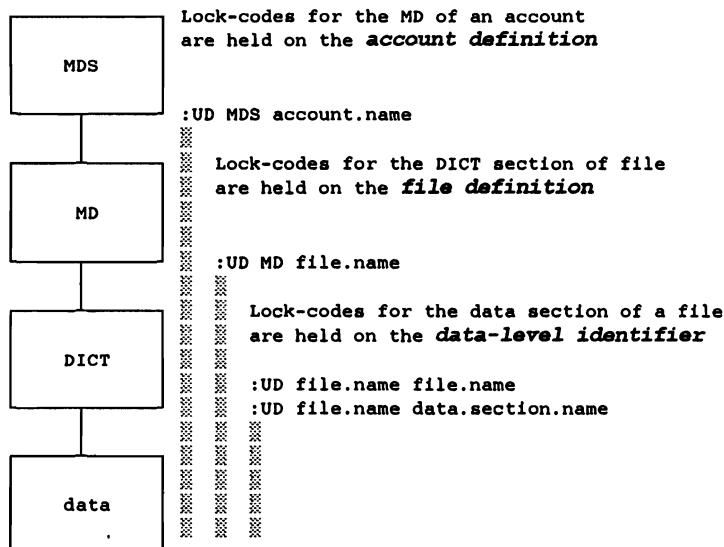


Figure 23.1: Lock-codes

Let us imagine that the owner of the WAGES account wishes to restrict access to the STATISTICS file and has placed the lock-code:

MONDAY

in the file definition for the STATISTICS file (held on the MD of the WAGES account). Now, in order to access this file from, say, the SALES account, a user with the user-id FRED logged on to the SALES account might attempt to use references such as:

```
LIST WAGES,STATISTICS,
UPDATE WAGES,STATISTICS, '1000'
```

or Fred might create a Q-pointer to the required file on the SALES MD and use this in a sequence such as:

```
SET-FILE WAGES STATISTICS
LIST QFILE
UPDATE QFILE '1000'
```

or:

```
SET-FILE WAGES STATISTICS MYSTATS
LIST MYSTATS
UPDATE MYSTATS '1000'
```

In order to be able successfully to access the STATISTICS file in this manner, the user FRED must hold (in the KEYS field of his USERS file) the matching key: MONDAY. If there is no such key, then the above LIST and UPDATE commands will be rejected by a message such as:

File 'STATISTICS' is access protected

and no action would be taken.

There are two lock-codes, known symbolically as RET-LOCK (or L/RET) and UPD-LOCK (or L/UPD). RET-LOCK is held in attribute 5 of a D-pointer item and will restrict attempts to *retrieve data from that file*; this includes such activities as using Access, reading, printing, and copying from the file. Some examples are shown in the table below. UPD-LOCK is held in attribute 6 of a D-pointer item and will restrict attempts to *update that file*; this includes such activities as writing and editing the file.

File lock-codes may be set and changed by updating the account definition, the file definition and/or the data-level identifier, as illustrated in Figure 23.1.

23.4 Logon procedures

When a user logs on to Advanced Pick, the sequence of events is:

- 1) The user-id and password are accepted and validated, and then ...
- 2) Any instructions specified in the MACRO – that is, attributes 9 onwards – of the USERS items are carried out, and then ...

If there is no LOGTO (or TO) command in the MACRO, then the name of the account (master dictionary) and password are accepted and validated; if a LOGTO command is included in the MACRO, then the user is logged to that account, and then ...

- 3) The MD is scanned for an item which has the same name (item-id) as the account or as the account synonym. If this is an executable item (such as a Proc or a macro or a menu or a catalogued program or a verb), then it will be invoked; if there is no such item, then the user is passed to the TCL prompt.

For example, if a user called FREDa logs to an account called WAGES, any executable item with the item-id WAGES on the MD will be invoked. If there is no such WAGES item (or there is a WAGES item but it is not executable), then the user passes to the TCL prompt. A command which has the same name as the account is called a **logon procedure**. A typical use of the logon procedure is:

- o To throw the users straight into an application system,
- o To ask for extra identification such as a further password or entry code,
- o To carry out a security check that the user / the port / the time of day are acceptable for work on that account.

The first of these is possibly the most common, with the logon procedure calling up a standard application program so as to confine the users to this program. Such a program would guide the users, prompt them, control them, and vet and restrict the activities which they perform. In this

manner, the users need not know about TCL commands or how to use the TCL language.

23.5 Closed systems

Even when they are thrown into a specific application, if the users press the <Break> key, they may be able to interrupt the processing and reach TCL level. In the situation where it is required to shield the users completely from TCL or to prevent them from using the TCL facilities, this use of the <Break> key is clearly undesirable. However, it can be arranged that any attempt to break out in this way will be ignored (by disabling the <Break> key off), or by forcing the user to return to the initial logon procedure. If you organise your processing in this manner, it is known as a **closed system**.

A logon procedure on the MD for the WAGES account might look like that in Figure 23.2 (a). A closed system is set up by creating an account definition on the MDS file, as in Figure 23.2 (b).

```
00  WAGES
01  N
02  BLOCK-PRINT WAGES ACCOUNT
03  RUN WAGES.BP MASTER.MENU
04  OFF
```

(a) A logon procedure

```
00  WAGES
01  D
02  599034
03  37
04
05
06
07
08  SYS2
09  R                                     <= OPTIONS field
10  10
```

(b) An MDS item

Figure 23.2: Logon procedures and closed systems

Note the R in attribute 9 – the OPTIONS field – of this item. It is this R – for **re-entrant** – in the account definition which prevents users of this account from reaching TCL by breaking out of the logon procedure. If a

user does press the <Break> key, the normal:

::

prompt (or possibly the debug prompt * or !) will appear. If the user enters END then control will be passed back to the logon procedure and not to TCL.

The use of **account synonyms** means that an account may be reached via several different *account names*, and that each synonym will have its own **logon procedure** and **system privileges** even though all users of the various synonyms are working on the same true account.

23.6 Other security features

Since the MD is the repository for all the commands and verbs which are available to the users of an account (and its synonyms), it is a simple matter to remove any verbs, such as:

- EDIT
- UPDATE
- MESSAGE / MSG

and any others which may jeopardise the security of the system or which are likely to be hazardous for the inexperienced user.

```
:listu

pib# user..... udate utime account..... mdate mtime location...

* 0 dm          08/10 10:00 dm          24/08 18:28 dp room
  1 mary        08/09 09:33 dm          24/08 16:33 programmers
  2 bert        08/09 08:15 develop    24/08 13:33 programmers
  3 samir       08/10 05:22 develop    24/08 14:45 manager

:who *

* 0 dm dm
  1 mary dm
  2 bert develop
  3 samir develop

:where

Ln  PCB      PIB  ABS  Stat  Rl & Return stack contents
   FID      Stat Bas
```

*000	00635C	FF10	000018	T	1	ws.where1:000	ws.whatwhere:236
065	000841	BF10	000018	P		sp.sleep:027	sp.spoolout:07A
073	006593	BF10	000018	P		pp.sched:02A	

Figure 23.3: LISTU / WHO / WHERE

There are a number of other tools at the System Manager's disposal to monitor what users are doing or have done in the past:

- o The LISTU command shows details of who is currently logged on to the system. The location is taken from the PIBS file. A typical report is shown in Figure 23.3.
- o The WHO command shows the user-id and account name for all current users. A typical report is shown in Figure 23.3.
- o The WHERE command shows details of who is doing what, listing the AP routines which each is currently using. Typical reports are shown in Figure 23.3.
- o The LOG-MSG command has the form:

LOG-MSG message

and can be embedded in sensitive macros, Procs and programs to log a specific message on the ERRORS file. The process records the date, time, port-number, user-id, account and the message, and can be inspected by a simple SORT ERRORS command.

- o If the **TCL stacker** is enabled, the TCL-STACK file holds a list of the TCL commands which each user has issued (repeated commands only appear once).
- o If the **capturing** feature is enabled, the CAPTCL file holds a record of every TCL command which each user has issued (repeated commands and null commands are recorded exactly as they were entered).

23.7 Summary

A variety of facilities are provided for controlling the users' access to the Advanced Pick system, for controlling and monitoring what they do, what they are doing and what they have done, and for controlling their access to the database.

24

Group format errors : GFEs

A **group format error** – or a **GFE** – is a serious error within the data recorded in a frame of the virtual memory system. The presence of a GFE may result in Advanced Pick being unable to reach and process data which lies in that frame and also in the group of frames which are linked on beyond the site of the error.

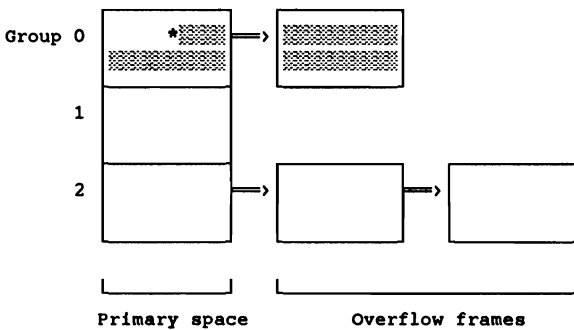


Figure 24.1: Groups and GFEs

It is called a **group** format error because, as we saw in Chapter 4, the frames occupied by a file are organised as a sequence of **groups**, each group consisting of a single frame or, if the group overflows, a chain of linked frames. When a new file is created, you will specify that it is to consist of a number of groups of frames; the file in Figure 24.1 has a modulo of 3 and therefore consists of three groups. As more and more items are added to the file, the primary file space occupied by one or more of the groups may overflow into additional frame(s) of disk space, as we see here. Now, if a GFE were to occur at the position indicated by the * in the first frame of group 0, then any data which lay in the shaded area of group 0 *beyond* the site of the GFE would be inaccessible to the normal Advanced Pick processing operations such as:

- the Editor,
- the Update Processor,
- the Access processor,
- the Basic READ/WRITE operations,
- the COPY processor,

and others. In fact, the only processing routines which *can* access all the

data in the affected frame(s) are the:

the CLEAR-FILE processor,
the DELETE-FILE processor,
the DUMP processor, and
the system debugger.

Note that *only the group in which the GFE occurs* is affected. The other groups in the file and any other parts of the system are not *directly* affected. I use the word *directly* because if the affected frame(s) hold items – such as D-pointers – which are used to access other parts of the system, then other areas may be *indirectly* affected.

Note also that the error affects only the *logical* meaning of the data. It is not a permanent *physical* error which will affect that frame indefinitely; such frames are recognised as faulty and ignored by Advanced Pick.

24.1 Nature of GFEs

In order to explain the nature of GFEs, we need to remind ourselves about the physical structure of the frames and the groups which make up a file. Let us imagine that group 0 of the our file starts at frame 8000 (the GROUP command would tell us this) and that it overflows into frame 9876; this is illustrated in Figure 24.2 (a). We could use the DUMP command to inspect the actual frames, as in (b).

The top line of each frame shows three pieces of information: the FID of this frame, the forward link (the FID of the frame into which this frame overflows, if any), and the backward link (the FID of the frame from which this frame overflows, if any). This linkage information is actually held in the final 24 bytes of the frame (or the last 48 bytes on systems which use 2048-byte frames). Look at the data in frame 8000. You can clearly identify the information relating to item 3333 on the file. You will recall from Chapter 4 that each item is preceded by an 8-byte control string (visible as `....c...` for item 3333) which tells Advanced Pick about the item: how big the item is, whether it is held directly or indirectly, whether or not it has been changed since the last file-save, and so on. You will also recall that each item is terminated by the end-of-item marker – shown here as the `^` followed by one or two `_` characters. There is also the end-of-group marker, shown here as the `_` character, after the data in the group.

In order for Advanced Pick to be able successfully to interpret the data in these frames, it must use three critical pieces of information:

- The **linkage information** must be correct, if it has to go from frame to frame.

- The **8-byte control field** must contain acceptable values for the length of the item, and
- The **control field** must contain an acceptable indication of whether the item is held directly or indirectly.
- The **expected length** (as taken from the control field) must tie up with the actual physical length (as indicated by the end-of-item marker).

As it attempts to read and interpret the frames in a group, Advanced Pick may find that:

- There is an invalid character in the linkage information which results in the system being unable to follow up any further frames in the group.
- There is any invalid character in the 8-byte control field, so that the processor does not know the expected length or the nature of the item.

In Figure 24.2 (b), you can see such an invalid control field on the line starting at byte 450 of frame 8000. This might have been caused by some electro-mechanical fault as the frame was being written to disk and would generate a GFE if we later attempted to read item 6000 or any of the following items.

- The position of the final end-of-item marker is inconsistent with the item length count.
- The final end-of-group marker is not the expected character.

If any of this data is corrupt, a group format error may be reported.

24.2 Causes of GFEs

Any of these situations may be caused by an electrical fault and/or a mechanical problem, such as:

- A power surge or power failure, or someone switching off or rebooting the system as Advanced Pick was writing a frame to disk.
- A hardware problem.
- Someone physically amending the modulo in a D-pointer.
- Someone using the system debugger and patching a frame incorrectly.
- A software problem, particularly in processes which utilise non-standard Assembly language routines.





(a) Group 0

```
:DUMP 8000 G
```

```
FID: 8000 : 0 9876 0 0
000 :....C...3333^CHAIR, WHITE, LAMINATE^84^MN/19/13^25:
050 :00^60^2000^9575^_....C...6666^SIDEBOARD, ORANGE, :
100 :OAK^88^LS/17/33^13000^15^1000^9579^_....?...4500^C:
150 :HAIR, RED, LAMINATE^18^LN/3/29^2500^60^1000^9565^_:
200 :....C...9000^SIDEBOARD, YELLOW, ASH^99^DN/14/70^13:
250 :000^15^1000^9539^_....?...4545^SETTEE, BLUE, MAPLE:
300 :^8^DN/60/39^1000^30^2000^9554^_....?...1500^CHAIR:
350 : , ORANGE, PINE^33^MN/6/13^2500^60^2000^9552^_....:
400 :C...3579^SIDEBOARD, GREY, MAPLE^6^LN/33/11^13000^1:
450 :5^1000^9543^_???????6000^SIDEBOARD, GOLD, OAK^0^:
500 :MN/3/66^13000^15^1000^9542^_....a...8763^SIDEBOARD:
550 : , MAROON, ASH^10]5]0^LS/33/4]LN/51/7]MN/3/40^13000:
600 :^15^1000^9561]9581]9571^_....A...3000^SIDEBOARD, :
650 :BLUE, ASH^58^MN/5/56^13000^15^2000^9537^_....=...:
700 :7500^CHAIR, BLACK, PINE^85^LS/4/82^2500^60^2000^95:
750 :77^_....[...4200^SETTEE, GOLD, ASH^55]10^XN/19/35]:
800 :LS/6/6^1000^30^1000]2000]3000]4000^9559]9565^_....:
850 :U...1200^DESK, GREY, ASH^16]32^LS/17/1]LS/17/2^560:
900 :0^30^1000]3000]2000^9538]9536^_....]...7245^BUREA:
950 :U, GOLD, ASH^55]10^LN/19/35]LN/6/6^10000^10^1000]2:
```

```
+ FID: 9876 : 0 0 8000 0
000 :000]3000]4000^9568]9542^_....=...9999^CHAIR, BLUE:
050 : , PINE^12^MN/16/69^2500^60^2000^9568^_....C...7777:
100 :^STOOL, YELLOW, LAMINATE^25^DN/7/23^1775^60^2000^9:
150 :582^_....=...1111^STOOLS, GREEN, PINE^63^DN/6/3^2:
200 :000^60^2000^9575^_....?...5500^TABLE, YELLOW, PINE:
250 :^15^LS/15/77^5000^25^2000^9549^_:
300 :_____:
```

(b) The contents of the frames

Figure 24.2: A frame dump

Any GFE is a cause for concern. If your system appears to be prone to GFEs, then you should call in the support staff from your hardware supplier and/or your software supplier.

There is a rare phenomenon known as a **soft GFE**. This is a situation which is reported as a GFE but which disappears a moment later and may be a result of one process attempting to read a piece of data at the exact moment that another process is writing the frame.

A GFE must not be confused with the harmless **frame fault** which is a term used to describe the situation in which a process is interrupted

momentarily as a new frame of data is brought in from disk.

Nor should you confuse a GFE with a physical disk error which may be the result of permanent physical damage to a part of the disk. Such a disk error renders a frame inaccessible to Advanced Pick and will normally be handled by the system itself.

24.3 Locating GFEs

GFEs are reported with the message:

```
*** GFE encountered
```

and the GFE handler will then allow you to take appropriate action, as we shall discuss shortly. Since Advanced Pick only senses a GFE when it needs to access data in the group which contains the error, a GFE may lie undetected for some time.

In order to search for GFEs, you must force the system to look at what you suspect to be the offending area. There are several ways in which you might do this:

- 1) To look for GFEs in a suspect file, you may use commands such as:

```
LIST STAFF WITH A1 NOPAGE
LIST ONLY STAFF WITH A1 NOPAGE
COUNT STAFF WITH A1
```

- 2) To look for GFEs in a suspect account:

```
CHECK-ACCOUNT
SAVE (FI
```

- 3) To look for GFEs throughout the system:

```
CHECK-FILES
SAVE (F
```

In all cases, any GFEs will be reported. If the process runs to completion without any errors being reported, then there are no GFEs on the file / account / system.

Unfortunately, the GFE handler does not show the actual frame where the GFE occurred. A solution to this can be found in the information logged on the ERRORS file. When a GFE is logged on the ERRORS file, the log item contains certain information which does not appear on the normal LIST ERRORS report. The following datanames are useful here:

```
DATE
TIME
```

GFE-FILE this shows the name of the file where the GFE occurred.

GFE-FID this shows the FID of the frame (in hexadecimal) and the location of the GFE within the frame.

LAST-TCL-ENTRY this shows the commands which were being processed when the GFE was detected.

and these can be used in your own Access reports to locate the GFE.

24.4 Fixing GFEs

When a GFE is detected, Advanced Pick calls up the GFE handler. This logs the situation on the ERRORS file, dumps the offending frame(s) to the ERRORS,GFE file, and displays the message:

```
*** GFE encountered
F=fix item / T=truncate group & quit / Q=just quit / <CR>=go to debugger
```

and waits for you to select any of the actions:

F this will make the best possible sense of the data up to the site of the GFE and then set an end-of-group marker to ignore the rest of the data in the group. A copy of the lost data is dumped to the ERRORS,GFE file.

T at the time of writing, there appears to be no practical difference between the effect of the F and the T responses.

Q to ignore the GFE, abandon the process and return to TCL.

<Return> to pass to the system debugger.

If you are in any doubt, you should select the Q option. This will give you time to think about what to do about the problem.

Any GFEs detected during a file-save / account-save will also be passed to the GFE handler, as we see in the following section.

If a user has M in the OPTIONS field of his/her user-id item on the USERS file, any GFEs detected by that user will automatically be fixed.

24.5 GFEs and file-saves

Since the file-save and account-save processes handle all the data on all the files which they backup, it is possible that GFEs may be detected during these processes.

The FILE-SAVE utility asks:

```
Bypass groups with GFE's?
```

to which you may reply:

- N if you want the process to stop at each GFE so that you may take appropriate action.
- Y if you want save all possible data without any user intervention. This will enable the process to run unattended.

If you enter N, indicating that you want to intercept each GFE, then, as each GFE is detected, the process will report:

```
*** GFE encountered
T=truncate group & continue / S=skip & continue / Q=just quit / <CR>=go to
  debugger ?
```

and wait for one of the responses:

- T will fix the GFE on disk and then continue to save any items which it can;
- S will leave the disk data unchanged and save any items which it can.

If you indicated that you want to bypass any GFEs, this is equivalent to entering the S option as each GFE is detected.

The same effect is produced by using the G option on the SAVE verb; this will report any GFEs encountered and then skip that group and continue.

In all cases, any errors are logged in the ERRORS file.

24.6 Recovering from GFEs

Since a GFE is caused by some inconsistency in the information recorded in a frame, it is (in principle) possible to recover from the error by patching the frame to the correct state. However, this is a fairly dangerous activity and quite likely to cause more problems than it solves. We look at the way in which the system debugger allows us to patch frames in Chapter 18.

By far the best – and safest – way to recover from any loss of data caused by GFEs is:

- o Get the GFE handler to fix the GFE.
- o Dump the offending frame(s) or inspect the dump which the GFE handler places on the ERRORS,GFE file to determine which items have been lost.
- o Use the SEL-RESTORE process to retrieve the lost item(s) from the latest file-save / account-save tape.

24.7 Summary

A GFE, a group format error, is a logical inconsistency in the data recorded in a frame of virtual memory. The presence of a GFE means that the information in the frame cannot be interpreted correctly and the items at and beyond the site of the GFE are inaccessible to most Advanced Pick processes. The GFE handler is automatically called up when a GFE is detected, and this gives you a number of options for fixing, removing, and clearing the problem.

Data lost as a result of a GFE should be recovered from a recent backup copy.

25

AP & the host operating system

In Chapter 1, we saw that Advanced Pick may be installed as an independent operating system running on a dedicated machine or in a dedicated partition; this is **AP/Native**.

The **AP/DOS** implementation and the **AP/Unix** implementations run as normal DOS and Unix processes alongside other DOS and Unix processes. In either case, DOS or Unix is known as the **host operating system**, and it is possible to communicate directly with the host operating system and to transfer data to and fro between Advanced Pick and the host.

25.1 Shelling out

From AP/DOS and AP/Unix, it is possible to issue commands directly to the host operating system, just as if they were TCL commands. This is achieved by prefixing the operating system command by the **!** character. For example, on an AP/DOS system, the command:

```
!DIR
```

will invoke the DOS command to display a report of the current directory. Any DOS command may be invoked in this way and the commands have their normal DOS form. The command:

```
!
```

alone will invoke the DOS shell and allow you to work directly at the DOS prompt line. The DOS EXIT command or the ^D character will pass control back to Advanced Pick. You can use any of . or DOS or PC or SHELL or SH or SHL as alternatives to the **!** identifier for DOS.

The effect is similar on AP/Unix systems. The Advanced Pick command:

```
!DIR
```

will invoke the Unix command to display a report of the current directory. Any Unix command may be invoked in this way and the commands have their normal Unix form. The command:

```
!
```

alone will invoke the Unix shell and allow you to work directly at the Unix prompt line. The Unix EXIT command or the ^D character will pass control back to Advanced Pick. You can SH as an alternative to the **!** identifier for Unix.

There are a number of TCL commands which invoke the equivalent Unix command directly. These include: CC / CPIO / ENV / GREP / LL / LS / PG / PWD / STTY / SU / TTY, and the command:

```
PSH unix.command > {file} item {(T}
```

or, if the Unix command contains any > characters:

```
PSH unix.command "> {file} item {(T}
```

will execute a Unix command and save the captured results as an item on an Advanced Pick file. If the file name is omitted, LST will be assumed. The T option displays the captured results.

The ENVIRON command allows you to manipulate the Unix shell environment variables directly from TCL.

25.2 Exporting to AP/DOS

The EXPORT verb is used to transfer data from Advanced Pick to the host operating system. For AP/DOS, this has the form:

```
EXPORT file.name item.list {(options)}
```

specifying the name of the Pick file and the item-id(s) of the items which are to be exported. The options include:

- C concatenate the items into one DOS file.
- E append an item delimiter to each item.
- I include the item-ids in the DOS file.
- K suppress the conversion of attribute-marks to the <Return> <Line Feed> sequence.
- N sort the items before exporting.
- O overwrite any existing DOS file.
- P send the output to the spooler.
- S use the Pick item-id(s) as the DOS file name(s). This is the default.
- T translate specific characters to new characters.

The process will then ask for the name of the DOS file to which the data is to be exported.

25.3 Exporting to AP/Unix

The EXPORT verb is used to transfer data from Advanced Pick to the host operating system. For AP/Unix, this has the form:

```
EXPORT file.name item.list {(options)}
```

specifying the name of the Pick file and the item-id(s) of the items which are to be exported. The options include:

- c suppress the conversion of attribute-marks to the <Return> <Line Feed> sequence.

- G allocate read/write permission to the group for the Unix file.
- I suppress the display of the item-id(s).
- L convert the Unix file name to lower-case.
- W allocate system-wide read/write permission for the Unix file.

The process will then ask for the name of the host file to which the data is to be exported.

25.4 Exporting from AP/Native to diskette/tape

AP/Native systems will export files to backing storage – diskette or tape – in a form which can be read on DOS systems. The data is copied from the Advanced Pick file and transferred to the *backing storage device which is currently attached*. The EXPORT command to do this has the form:

```
EXPORT file.name item.list {(options)}
```

specifying the name of the Pick file and item-ids of the items which are to be exported. The options include:

- C concatenate the items into one DOS file.
- E append an item delimiter to each item.
- I include the item-ids in the DOS file.
- K suppress the conversion of attribute-marks to the <Return> <Line Feed> sequence.
- N sort the items before exporting.
- O overwrite any existing DOS file.
- P send the output to the spooler.
- S use the Pick item-id(s) as the DOS file name(s). This is the default.
- T translate specific characters to new characters.

The process will then ask for the name of the host file to which the data is to be copied on the diskette, and this will be entered in the DOS form but **without a drive letter**.

25.5 Importing from AP/DOS

The IMPORT verb is used to transfer data to Advanced Pick from the host operating system. For AP/DOS, this has the form:

```
IMPORT dos.file.name {(option)}
```

specifying the name of the DOS file – including the drive and directory path – which is to be imported. The options include:

- o overwrite the item if it already exists in the Pick file.

The process will then ask for the name of the Advanced Pick file and the item-id to which the data is to be copied.

25.6 Importing from AP/Unix

The **IMPORT** verb is used to transfer data to Advanced Pick from the host operating system. For AP/Unix, this has the form:

```
IMPORT pick.file.name item.list {(options)}
```

specifying the name of the Pick file and the item-id(s) of the items to which the Unix files are to be imported. The options include:

- E***n* expand any Unix tabulations to *n* spaces. If *n* is omitted, 4 is assumed.
- I** suppress the display of item-id(s).
- L** convert the Unix file name(s) to lower-case.
- O** overwrite the item if it already exists in the Pick file.

The process will then ask for the name of the Unix files from which the data is to be imported.

25.7 Importing to AP/Native from diskette/tape

AP/Native systems will import files from a DOS diskette or the DOS partition into an Advanced Pick file. The **IMPORT** command to do this has the form:

```
IMPORT dos.file.name {(options)}
```

and the options include:

- O** overwrite any existing Pick file.

The process will then ask for the name of Advanced Pick file and item-id to which the file is to be imported.

25.8 Summary

On those implementations of Advanced Pick which run within DOS or Unix environments, it is possible to issue commands to the host operating system directly from AP TCL, and you may transfer data between Advanced Pick and the host operating system or a diskette/tape in DOS format.

SQL : the structured query language

Earlier in the book, we looked closely at Access, the enquiry language which is a fundamental part of the Pick system and the Advanced Pick system. Advanced Pick release 6.1 and later include the standard SQL – the structured query language – to the armoury of Advanced Pick; this makes the entire Advanced Pick database accessible to anyone who is familiar with SQL and extends the power of Pick to other environments.

The dialect of SQL offered on Advanced Pick is based on current industry standards for the language.

SQL can used in an **interactive** mode, typing SQL commands directly, or SQL commands may be embedded in Basic programs and executed from there.

26.1 The SQL jargon

The terms used in SQL are slightly different from those used in other areas of Advanced Pick. The table in Figure 26.1 shows these.

SQL	Pick	RDB
Table	File	Relation
Row	Item	Tuple
Column	Attribute	Attribute
Value	Value	Value
Single nested table	Multivalue	Relation-valued attribute
Double nested table	Subvalue	Relation-valued attribute

26.2 CREATE TABLE

26.3 SELECT

26.4 Indexes

26.5 Views

26.6 Summary

A

Glossary

Access: The enquiry language which is available on the Advanced Pick system and which allows you to make enquiries and to produce reports by typing in an English sentence.

Account: A set of files which are grouped together and accessible to anyone who logs to that account.

Account definition: An item on the MDS file which defines the MD for a particular account. The account name is the item-id of account definition.

Account synonym: An item on the MDS file which allows an account to be identified by an alternative name. The account synonym may have different lock-codes and different privileges from the main account.

Account restore: The process of recovering the entire contents of the files of a specific account from a copy dumped to backing store (tape / diskette) by the file save process or the account save process.

Account save: The process of dumping the contents of a specific account to backing storage.

Advanced Pick / AP: The latest version of the Pick system. There are several implementations of AP: **AP/DOS** which runs as a DOS program entirely within the DOS environment; **AP/Unix** which runs as a Unix process within the Unix environment; **AP/Native** which runs in a dedicated partition on 286 or higher systems.

AP: Abbreviation for **Advanced Pick**.

AP/DOS: An implementation of Advanced Pick which runs as a DOS process within a normal DOS partition.

AP/Native: An implementation of Advanced Pick which runs in a dedicated partition on a 286 or higher computer system.

AP/Unix: An implementation of Advanced Pick which runs as a process within the Unix environment.

Attribute: The Pick terminology for a field or a data field of a record. Each field of a Pick data item is known as an attribute. Within the item, the attributes are separated by the attribute mark (ASCII character 254). Each attribute contains any number of **values** separated by value-marks. Each value contains any number of **subvalues** separated by subvalue-marks. Attributes, values, and subvalues are of variable length, each being terminated by the appropriate field-separator. The various

data fields — the attributes, values, and subvalues — are identified by their sequential position within the item. A field which contains no data is known as a null field. You should distinguish between a null field (which contains no data whatsoever) and a field which contains only spaces. When you are entering information at the keyboard, you will normally represent a null field just by hitting the <Return> key. Null fields are represented only by the associated (following) field-separator — attribute-mark, value-mark, or subvalue-mark.

Attribute definition: An item on the dictionary section of a file defining a dataname which can be used in Access sentences. The dataname is the item-id of attribute definition.

Basic: The language which is used to write programs for the computer. There are many such programming languages available. The standard Pick system can handle programs written in the Basic language and the Proc language.

Basic debugger: A standard piece of software which allows the programmer to identify and locate errors in a Basic program.

Break condition: A state of the data or the execution of a program at which the execution is to be interrupted and control passed to the debugger. Break conditions are set by means of the B command, and the program execution will be interrupted when any one is reached.

Break-point: A point at which the normal processing action of a program is interrupted — either by accidental error or by reaching a break condition set by the programmer — and the Basic debugger is called into action by the Pick system.

Break-point table: That part of the trace table which holds details of the current break conditions.

Bridging: Advanced Pick facility whereby the integrity between two files may be maintained as items are added, changed and deleted from a file.

Command: An instruction given to the Pick system when at the TCL level.

Cruising: Advanced Pick facility whereby the user may browse through the items in a file for which an index is being maintained.

Data-level identifier: An item on the DICT section of a file which defines one of the data sections belonging to that file. The data section name is the item-id of data-level identifier.

Dataname: The name by which each data field is identified in an Access sentence. There must be an attribute definition for each such dataname.

Direct item: An item which is held wholly within the primary file space and/or overflow space of its group. Contrasts with **Indirect item**.

DM: The Data Manager account of an Advanced Pick implementation. This is equivalent to the SYSPROG account on R83 Pick.

DM account: The Data Manager account of Advanced Pick which is provided for the system management facilities. This is equivalent to the SYSPROG account on R83 Pick.

Double-clutching: Advanced Pick facility whereby the user, whilst processing one file, may browse through the items in a second file for specific values there.

Editor: A piece of software which is provided to create, change and delete records on files. The Pick Editor is a line-editor for use in creating and maintaining items on Pick files.

FID: Abbreviation for Frame Identifier. The number which identifies a frame within the Advanced Pick virtual memory.

File: A collection of items holding data of a similar nature. This is identical to the standard use of the term file within conventional data processing. A file may contain any number of items (or records).

File definition: An item on the MD which defines a particular file belonging to that account. The file name is the item-id of file definition.

File restore: The process of recovering the entire system from a copy dumped to backing store by the file save process. The files on the hard disk are completely reorganised as the process takes place.

File save: The process of dumping the contents of the entire system to backing storage (tape or diskette). File saves are performed for security purposes and allow and lost or corrupt items to be recovered to their state at the time the file save was performed.

File synonym: An item on the MD which allows a file on this or another account to be identified by an alternative name.

Frame: The physical unit in which data is moved between the hard disk and the Advanced Pick virtual memory.

GraPick: A graphical user interface for the Advanced Pick system.

In-line item: An alternative to the term **direct item**.

Index: An operational file which enables items on the main file to be located via some key(s) other than the unique item-id. Open Architecture and Advanced Pick offer an indexing mechanism whereby one or more indexes may be maintained whenever items are added, changed or deleted from a file.

Indirect item: An item which is held in the extended file space with just a pointer to the extended file space address being held in the primary file space and/or overflow space of its group. Contrasts with **direct item**.

Interactive debugger: Another name for the system debugger.

Item: The Pick term for a data record. Each file consists of any number of items. Each item consists of the **item-id** which identifies the item, followed by none, one or more **attributes** separated by attribute-marks.

Level pushing: The Advanced Pick facility whereby the current process may be interrupted (by hitting the <Break> key) and a second process invoked. In turn, this second process may be interrupted and a further process invoked. As each low-level process terminates, control is returned to the process at the previous level.

Macro: A list of attribute names held in attribute 15 of a file definition. This list of names is used in Access sentences and Update Processor commands if no other list of names is specified in the command. A macro may also appear in an attribute definition and specifies the names which are to be used when this field is used to zoom to another item.

Macros: Advanced Pick offers macros as a means of saving and invoking a stream of one or more TCL commands. Unlike Procs, it is not possible to tailor macros at run-time. A macro is held in a macro definition on the MD.

Master dictionary: The file which holds all the verb definitions and file definitions which are available to an account.

MD: Abbreviation for **Master Dictionary**.

MDS file: The standard file which holds details of all the accounts on the Advanced Pick system. For each account, the MDS file holds an account definition for the MD of that account.

Menu: A means of invoking a TCL command or Access sentence by selecting from a displayed list of available commands and sentences.

Multivalue: Each attribute in a Pick item may consist of a none, one or more multivalues.

OA: Abbreviation for **Open Architecture**.

Object program: The executable version of a program which is derived from the Basic **source program** by the **compiler**.

Open Architecture: A version of the Pick system between R83 Pick and Advanced Pick.

Out-of-line item: An alternative to the term **indirect item**.

Output processor: The text processing utility of Advanced Pick which is provided for the production of printed material and documents. It extends the features of Runoff and allows control commands to be embedded within the text lines.

Password: A string of characters which must be entered with a user-id (or a password) when logging on to the system.

Phrase: A section of an Access sentence which has been set up as an item on the dictionary of a file.

Proc: A sequence of instructions – typically for the construction of an Access sentence or a TCL command – which has been saved under a single name so that it can easily be recalled in future. The term is also used to refer to the language which is used to write such processing routines.

Q-pointer: A synonym item — such as an account synonym or a file synonym — which allows an account or a file to be identified by an alternative name.

R83 Pick: The original version of the Pick system and the predecessor of Open Architecture and Advanced Pick. Most Pick-like systems are based upon R83 Pick.

Runoff: The standard text processing software of the R83 Pick and Advanced Pick systems. Although still available, it has been superseded by the Output Processor on AP.

Selection criterion: That part of an Access sentence which starts with the word WITH or IF and which indicates which items are to be included in the report.

Selective restore: The operation in which one or more items are restored to a file by taking the data from a file save / account save held on backing storage. Usually done when items on the live system have been (accidentally or otherwise) removed.

Sentence: A command in the Access enquiry language. An Access sentence must begin with one of the Access verbs.

Sort specification: That part of an Access sentence which starts with one of the keywords BY / BY-DSND / BY-EXP / BY-EXP-DSND and which controls the sequence in which the items of the report are to be sorted.

Source program: A sequence of Basic statements written by a programmer in order to perform some particular process. Before it can be executed, a source program must be converted into the equivalent object program by the compiler.

Stacker: A standard piece of software which allows TCL commands to be saved, changed and maintained, recalled and re-issued.

Subvalue: Any value of an item may hold zero, one or more separate subvalues. These are held within the value and separated by the subvalue mark (ASCII character 252). In general, subvalues are not used to store data in an application system.

Symbol table: A list containing references to all the variables, labels and subroutine calls used in a Basic program, and which the Basic run-time processor uses to identify the location of each object which it uses. The table is produced by the Basic compiler and (normally) held with the object program. It is particularly useful in conjunction with that feature of the Basic debugger which allows the programmer to interrogate (and change) the contents of the program variables, identifying them by name.

Symbolic debugger: Another name for the Basic debugger.

Synonym: An item — such as an account synonym / file synonym — which allows an account / a file to be identified by an alternative name. The alternative term **Q-pointer** is also used.

System debugger: A standard tool for use by systems programmers in identifying, locating and correcting errors in an assembly language process. Its main application by end-users is in inspecting and changing parts of the virtual memory system.

SYSTEM file: The standard file which holds details of all the accounts on the R83 Pick system. This is the equivalent of the MDS file on Advanced Pick.

System privileges: A code SYS0 / SYS1 / SYS2 in a user definition or an account definition which controls the scope of the actions which that user / account is able to perform.

TCL: The Terminal Control Language which is used as the fundamental means of communicating with the Pick system. The language consists of a series of commands typed in by the user or issued by a Proc or a Basic program. Each TCL command starts with a verb possibly followed by parameters and/or options for that verb.

Trace table: A table used by the Basic debugger holding a list of the names of up to six variables whose contents are to be displayed automatically at each break-point, and a list of up to four conditions which are to cause an interrupt.

Update processor: The Update Processor of Advanced Pick performs a number of roles: it is a general purpose editor, superseding the EDIT verb, and it is an application tool, allowing the maintenance of the data items on a file to be controlled by the attribute definitions on the data dictionary of the file.

User definition: An item on the USERS file which defines each individual user who is authorised to use the Advanced Pick system. The user-id is the item-id of the user definition.

User-id: The code by which each individual user identifies himself/herself to the Advanced Pick system.

USERS file: The standard file which holds details of all the authorised users on the Advanced Pick system. For each individual user, the USERS file holds an identifying item.

Value: Any attribute may comprise several values separated by the value-mark.

Zooming: Advanced Pick facility whereby the user, whilst processing one file, may jump across and browse through the items in a second file.

B

Output processor commands

The following Output Processor commands and abbreviated forms are available.

*		
APPENDIX / APX	ITALICS / IT	SUBSCRIPT / SUB
BEGIN PAGE / BP	JUSTIFY / J	SUPERSCRIPIT / SUP
BLOCK CENTER / BC	LEFT MARGIN / LM	TAB FILL / TF
BOLDFACE / BF	LINE LENGTH / LL	TAB LEFT / TL
BOX	LINE PRINTER / LP	TAB RIGHT / TR
BREAK / BK	LOWERCASE / LC	TAB RIGHTM / TRM
CAP SENTENCES / CS	LPI	TAB SET / TS
CENTER / C	MACRO FILE / MF	TABLE / TBL
CHAIN	NOFILL / NF	TCL
CHAPTER / CH	OVERCHAR / OC	TCL BOX / TCLBOX
CHAR	PAGE NUMBER / PN	TEST PAGE / TP
COL	PAGING / PG	TOC HEADING / TCH
COLUMNS SET / COLUMNS	PAPER LENGTH / PL	UNDERLINE / UL
CRT	PARAGRAPH / P	UNDERLINE WORDS / UW
CURSOR	PREFACE / PF	UPPERCASE / UC
DATE	PREFIX PAGE / PP	VARIABLE COLUMNS / VCOL
DEFAULT / DF	PRINT INDEX / PI	VMI
EM	PRINT PTOC / PPTOC	X
END CASE / EC	PRINT TOC / PTOC	XBLOCK CENTER / XBC
FIGURE / FIG	PROMPT / PRM	XBOLDFACE / XBF
FILL	READ / R	XBOX
FONT / F	READNEXT / RN	XCAP SENTENCES / XCS
FOOTING / FT	READNEXT NULL / RNN	XCOLUMNS
GOHANGING TAB / GHT	RESET / RST	XHILITE / XHI
HANGING TAB / HT	RESET PRINTER / RSTPTR	XHYPHENATE / XHY
HEADING / HD	RIGHT MARGIN / RM	XINDEX / XIX
HILITE / HI	SAVE CONTENTS / SC	XITALICS / XIT
HYPHENATE / HY	SAVE INDEX / SI	XJUSTIFY / XJ
INDENT / I	SECTION / SEC	XPAGING / XPG
INDENT MARGIN / IM	SET CHAPTER / SCH	XPARAGRAPH / XP
INDENT RMARGIN / IRM	SET SECTION / SS	XPREFACE / XPF
INDEX / IX	SKIP / SK	XPREFIX PAGE / XPP
INDEX HEADING / IXH	SPACE / SP	XSUBSCRIPT / XSUB
INPUT / IN	SPACING / SPG	XSUPERSCRIPT / XSUP
	STANDARD / STD	XUNDERLINE / XUW/XUL

C

Runoff commands

The following Runoff commands and abbreviated forms are available.

BEGIN PAGE / BP
BOX
BREAK / B
CAPITALIZE SENTENCES / CS
CENTER / C
CHAIN
CHAPTER
CONTENTS
CRT
END CASE / EC
FILL / F
FOOTING
HEADING
HILITE
INDENT / I
INDENT MARGIN / IM
INDEX
INPUT
JUSTIFY / J
LEFT MARGIN
LINE LENGTH
LOWER CASE / LC
LPTR
LX

NOCAPITALIZE SENTENCES / NCS
NOFILL / NF
NOJUSTIFY / NJ
NOPARAGRAPH
PAGE NUMBER
PAPER LENGTH
PARAGRAPH / P
PFILE
PRINT
PRINT INDEX
READ
READNEXT
SAVE CONTENTS)
SAVE INDEX)
SECTION
SET TABS
SKIP / SK
SPACE / SP
SPACING
STANDARD
TCL
TEST PAGE / TP
UPPER CASE / UC

D

Heading / footing options

The following options are generally available with the headings and footings used in Access sentences, Basic programs, the Output processor and other areas of Advanced Pick:

Axx to output the value of dataname xx.

' to output a single apostrophe, as mentioned below.

B{n} to output the BREAK-ON or ROLL-ON value left-justified in a field *n* characters wide. The 'B' option must be used in the BREAK-ON clause and also in the HEADING (and/or the FOOTING).

C to output the text in the centre of the line.

Cn to centre the following text within a field of *n* characters.

D to output the date in the format: 01 Jan 1999

DU to output the date, as for D, except that the date of a long report will be changed if it runs over midnight.

F{n} to output the file-name left-justified in a field of *n* spaces.

I{n} to output the item-id left-justified in a field of *n* spaces.

J to right-justify the following text.

L to start a new line.

N to suppress the pause at the end-of-page when displaying on the screen.

P to output the page-number right-justified in a field of four spaces.

PN to output the page-number with no surrounding spaces.

R to output the page-number in Roman numerals.

S to output subsequent text in italics.

T to output the date and time.

TU to output the date, as for T, except that the date of a long report will be changed if it runs over midnight.

U to underline subsequent text.

V to output subsequent text in bold-face.

- x to combine the effects of XS, XU and XV in disabling all special output effects.
- xc to cancel centering.
- xs to disable italic output which was invoked by the S option.
- xu to disable underlined output which was invoked by the U option.
- xv to disable bold-face output which was invoked by the V option.

If you wish to include an apostrophe in a heading / footing, you will represent this by two apostrophes:

..

For example, you might use the heading:

```
HEADING "'C'*** Kathy O'Donnell's report **'LL'"
```

to output:

```
** Kathy O'Donnell's report **
```

followed by two blank lines at the top-centre of every page of the report.

If there are two or more adjacent codes, you will enclose them in one set of apostrophes:

```
HEADING "'C'*** Sales Report **'LC'-----'L'"
FOOTING "'LLC'Produced 'T' ----- Page 'PN'"
```

E

Basic statements

The following statements are available in the version of Basic offered on Advanced Pick. Full details can be found in EPick.

!	GO / GOTO / GO TO	PROCWRITE
\$INCLUDE	GOSUB	PROGRAM
\$INSERT	HEADING	PROMPT
*	IF	READ
ABORT	IN	READNEXT
BEGIN CASE	INCLUDE	READT
BREAK	INPUT	READTL
CALL	INPUT @	READTX
CAPTURING	INPUTERR	READU
CASE	INPUTNULL	READV
CASING	INPUTTRAP	READVU
CHAIN	INS	RELEASE
CLEAR	KEY	REM
CLEARFILE	LET	REPLACE
CLOSE	LOCATE	RETURN
COMMON	LOCK	REWIND
COMMON /com/	LOOP / REPEAT	ROOT
CONVERT	MAT	RQM
CRT	MATBUILD	SELECT
DATA	MATPARSE	SEND
DEBUG	MATREAD	SLEEP
DEL	MATREADU	STOP
DELETE	MATWRITE	SUBROUTINE
DIMENSION	MATWRITEU	TCL
ECHO	NEXT	TCLREAD
END	NULL	UNLOCK
ENTER	ON ... GOSUB	WEOF
EQUATE	ON ... GOTO	WRITE
ERROR	OPEN	WRITET
EXECUTE	OUT	WRITEU
EXIT	PAGE	WRITEV
FILE	PRECISION	WRITEVU
FOOTING	PRINT	
FOR / NEXT	PRINTER	
GET	PROCREAD	

F

Basic functions

The following Basic functions are available on Advanced Pick.

@	COUNT	FOLD	OCCURS	SPACE
ABS	DATE	ICONV	OCNV	SQRT
ACCESS	DCOUNT	INDEX	PWR	STR
ALPHA	DELETE	INSERT	REM	SUM
ASCII	DTX	INT	REPLACE	SYSTEM
ASSIGNED	EBCDIC	LEN	RND	TAN
CHAR	ERROR	LN	SEQ	TIME
COL1	EXP	MOD	SIN	TIMEDATE
COL2	EXTRACT	NOT	SORT	TRIM
COS	FIELD	NUM	SOUNDEX	XTD

AP/DOS and AP/Unix also offer:

%ACCEPT	%FDOPEN	%GETHOSTID	%MEMXCPY	%SETFLUSH
%ALARM	%FGETC	%GETPGRP	%OPEN	%SHMAT
%BIND	%FGETS	%GETPID	%PAUSE	%SHMDT
%CHDIR	%FOPEN	%GETPPID	%PCLOSE	%SHMGET
%CHMOD	%FPRINTF	%IOCTL	%PGETPID	%SOCKET
%CHOWN	%FPUTC	%KILL	%POPEN	%TTYNAME
%CLOSE	%FPUTS	%LISTEN	%PUTENV	%UNLINK
%CONNECT	%FREE	%LSEEK	%RDHEX	%WAIT
%CREAT	%FREOPEN	%MALLOC	%READ	%WHEX
%DUP	%FSIZE	%MEMCCPY	%SEMCTL	%WRITE
%FCLOSE	%GETENV	%MEMCPY	%SEMGET	

If you are migrating from a system other than R83 Pick, you should note that the following Basic functions are not available on Advanced Pick:

ATAN	CRC	FFLT	KEYIDS	SERIAL
BCC	DECRYPT	FIELDSTORE	KEYSTRUCT	SMUL
BITAND	DIR	FMT	LINESTATUS	SPOOLER
BITCHANGE	DIRLIST	FMUL	LOCATE	SQUOTE
BITCHECK	DQUOTE	FSUB	MATCHFIELD	SSUB
BITLOAD	DRIVE	GET	MATUNPARSE	STATUS
BITNOT	ENCRYPT	GETBREAK	MAXIMUM	SUMMATION
BITOR	EOF	GETECHO	MINIMUM	SWAP
BITRESET	EREPLACE	GETMSG	NEG	TA
BITSET	ERRTEXT	GETPRINTER	QUOTE	TRANSQUERY
BITXOR	EXCHANGE	GETPROMPT	REUSE	TRIMB
CALCULATE	FADD	GROUP	SADD	TRIMF
CHANGE	FCMP	HASH	SCMP	UNASSIGNED
CHECKSUM	FDIV	INMAT	SDIV	USERTEXT
CONVERT	FFIX	INP	SENTENCE	XLATE

G

Proc statements

The following statements are available in the version of the Proc language offered on Advanced Pick. Full details can be found in EPick.

A{s}{n}{.m}	IF {#} S	RO
B	IHx	SP
BO	IN{p}	SS
C	IP{B}{p}{b}	ST OFF / ON
D{n}{.m}{+}	IP{p}	STOFF
F	IS{p}	STON
G n	IT	Sn
GO n	O	T
GOTO n	P	Uxxxx
Hx	PH	X{x}
H{x}<	PP	(f i)
H{x}<<	PQ	+n
IF A{n}{.m}	PW	-n
IF {#} A{n}	PX	[]
IF {#} E	RI{n}	[f i]

If you are migrating from a system other than R83 Pick, you should note that the following Proc statements are not available on Advanced Pick:

F-C / F-CLEAR	GO F	IT{C}{A}
F-D / F-DELETE	GOSUB n	L
F-F / F-FREE	Hb	M / MARK
F-O / F-OPEN	IBHb	MV
F-R / F-READ	IBHb:oconv;	MVA
F-U / F-UREAD	IBHb:iconv;	MVD
F-W / F-WRITE	IBHx	PLn
F;	IBN{p}	PQN
FB	IBP{p}{b}	Q
FBU	IFN	RSUB
GO A{n}	IH\	RTN
GO B	IH \	TR ON / OFF

H

ASCII characters

This chart shows the decimal and hexadecimal equivalents of the ASCII character set.

Dec	Hex	ASCII	Dec	Hex	ASCII	Dec	Hex	ASCII
0	0	NUL	41	29	)	82	52	R
1	1	SOH	42	2A	*	83	53	S
2	2	STX	43	2B	+	84	54	T
3	3	ETX	44	2C	,	85	55	U
4	4	EOT	45	2D	-	86	56	V
5	5	ENQ	46	2E	.	87	57	W
6	6	ACK	47	2F	/	88	58	X
7	7	BEL	48	30	0	89	59	Y
8	8	BS	49	31	1	90	5A	Z
9	9	HT	50	32	2	91	5B	[
10	A	LF	51	33	3	92	5C	\
11	B	VT	52	34	4	93	5D	]
12	C	FF	53	35	5	94	5E	^
13	D	CR	54	36	6	95	5F	_
14	E	SO	55	37	7	96	60	`
15	F	SI	56	38	8	97	61	a
16	10	DLE	57	39	9	98	62	b
17	11	DC1	58	3A	:	99	63	c
18	12	DC2	59	3B	;	100	64	d
19	13	DC3	60	3C	<	101	65	e
20	14	DC4	61	3D	=	102	66	f
21	15	NAK	62	3E	>	103	67	g
22	16	SYN	63	3F	?	104	68	h
23	17	ETB	64	40	@	105	69	i
24	18	CAN	65	41	A	106	6A	j
25	19	EM	66	42	B	107	6B	k
26	1A	SUB	67	43	C	108	6C	l
27	1B	ESC	68	44	D	109	6D	m
28	1C	FS	69	45	E	110	6E	n
29	1D	GS	70	46	F	111	6F	o
30	1E	RS	71	47	G	112	70	p
31	1F	US	72	48	H	113	71	q
32	20	Space	73	49	I	114	72	r
33	21	!	74	4A	J	115	73	s
34	22	"	75	4B	K	116	74	t
35	23	#	76	4C	L	117	75	u
36	24	\$	77	4D	M	118	76	v
37	25	%	78	4E	N	119	77	w
38	26	&	79	4F	O	120	78	x
39	27	'	80	50	P	121	79	y
40	28	(	81	51	Q	122	7A	z

123	7B	{	168	A8		212	D4	
124	7C		169	A9		213	D5	
125	7D	}	170	AA		214	D6	
126	7E	~	171	AB		215	D7	
127	7F	DEL	172	AC		216	D8	
128	80		173	AD		217	D9	
129	81		174	AE		218	DA	
130	82		175	AF		219	DB	
131	83		176	B0		220	DC	
132	84		177	B1		221	DD	
133	85		178	B2		222	DE	
134	86		179	B3		223	DF	
135	87		180	B4		224	E0	
136	88		181	B5		225	E1	
137	89		182	B6		226	E2	
138	8A		183	B7		227	E3	
139	8B		184	B8		228	E4	
140	8C		185	B9		229	E5	
141	8D		186	BA		230	E6	
142	8E		187	BB		231	E7	
143	8F		188	BC		232	E8	
144	90		189	BD		233	E9	
145	91		190	BE		234	EA	
146	92		191	BF		235	EB	
147	93		192	C0		236	EC	
148	94		193	C1		237	ED	
149	95		194	C2		238	EE	
150	96		195	C3		239	EF	
151	97		196	C4		240	FO	
152	98		197	C5		241	F1	
153	99		198	C6		242	F2	
154	9A		199	C7		243	F3	
155	9B		200	C8		244	F4	
156	9C		201	C9		245	F5	
157	9D		202	CA		246	F6	
158	9E		203	CB		247	F7	
159	9F		204	CC		248	F8	
160	A0		205	CD		249	F9	
161	A1		206	CE		250	FA	
162	A2		207	CF		251	FB	
163	A3		208	D0		252	FC	SVM
164	A4		209	D1		253	FD	VM
165	A5		210	D2		254	FE	AM
166	A6		211	D3		255	FF	SM
167	A7							

The characters SVM (the **subvalue-mark**), VM (the **value-mark**), AM (the **attribute-mark**) and SM (the **segment-mark**) are the field and data separators used by the Pick system.

I

Advanced Pick <Ctrl> keys

This table presents the <Ctrl> keys which are used with the various Advanced Pick tools. Here and throughout the book, the notation ^D represents <Ctrl>D and so on.

Key	TCL stacker
^D	Go to the next command in the stack
^F	Go to the previous command in the stack
^X	Abandon the displayed command
xx^A	Search for most recent command containing string xx
^A	Repeat the last search

^G	Move the cursor to the end of the displayed command
^J	Move the cursor one character to the left
^K	Move the cursor one character to the right
^T	Move the cursor to the beginning of the command
^U	Move the cursor one word to the right
^Y	Move the cursor one word to the left

	General text editing
^E	Delete from cursor position to the end of the sentence
^H	Overwrite character to the left with a space
^L	Delete the current character
^O	Delete to the end of a word
^R	Toggle between the overwrite/insert modes
^V	Insert a value-mark
^W	Insert one space
^ZC	Display the column position mask
^ZD	Delete from the cursor position to the end of the item
^ZO	Delete to the end of the paragraph/attribute
^ZT	Set tab stops
^ZZ	Undo the last change

	General cursor movement
<code>^B</code>	Move the cursor up one line
<code>^D</code>	Move the cursor one sentence to the left
<code>^F</code>	Move the cursor one sentence to the right
<code>^G</code>	Move the cursor to the end of paragraph/attribute
<code>^I</code>	Move the cursor right to the next tab stop
<code>^J</code>	Move the cursor one character to the left
<code>^K</code>	Move the cursor one character to the right
<code>^M</code>	Move the cursor to the next paragraph/attribute
<code>^N</code>	Move the cursor down one line
<code>^P</code>	Move the cursor down one screen (prestored command)
<code>^T</code>	Move the cursor to the top of the item
<code>^U</code>	Move the cursor one word to the right
<code>^Y</code>	Move the cursor one word to the left
<code>^Z9</code>	Move the cursor to attribute/line 9 of the item
<code>^ZA</code>	Redisplay the screen with the current line at the top
<code>^ZB</code>	Redisplay the screen with current line at the bottom
<code>^ZE</code>	Move the cursor to the end of the item
<code>^ZG</code>	Move the cursor to the end of paragraph/attribute
<code>^ZH</code>	Redisplay with current bottom line at line 11 of the screen and and the cursor at line 12
<code>^ZN</code>	Move the cursor down one screen
<code>^ZP</code>	Redisplay the screen
<code>^ZQ</code>	Redisplay with current bottom line at line 11 of the screen and the cursor at line 18
<code>^ZY</code>	Move the cursor up one screen

	General filing
<code>^XB</code>	Return to the previous item
<code>^XC</code>	File, compile and catalog the (program) item
<code>^XE</code>	Abandon the item without filing
<code>^XF</code>	File the item
<code>^XI</code>	Rename and file the item
<code>^KK</code>	Abandon the item(s) and return to TCL
<code>^XN</code>	Exit item and go to a new item
<code>^XO</code>	Exit and delete the item
<code>^XP</code>	File and print the item
<code>^XR</code>	File, compile and run the (program) item
<code>^XR</code>	File and invoke the (macro/menu/Proc) item
<code>^XS</code>	Save the item and continue
<code>^XX</code>	Abandon the item without filing

	General Update Processor functions
<code>^D</code>	Cruise to the previous indexed item and edit
<code>^E</code>	Get the next sequential index
<code>^F</code>	Cruise to the indexed item and edit that item
<code>^G</code>	Zoom to another file
<code>^U</code>	Get the next indexed item
<code>^Y</code>	Get the previous indexed item

	Cut and paste
<code>^CC</code>	Mark the end of a text block
<code>^CD</code>	Mark the start of the text block for deletion
<code>^CL</code>	Mark the start of the text block for copying
<code>^CP</code>	Paste in marked block
<code>^CRfi^M</code>	Read text from a file
<code>^CWfi^M</code>	Write current paste block to a file

	Prestore command buffer functions
<code>^P</code>	Execute command(s) in the prestore buffer
<code>^ZL</code>	Amend command(s) in the prestore buffer
<code>^ZRfi^M</code>	Read the prestore buffer from a file
<code>^ZRfi^P</code>	Read the prestore buffer from a file and execute
<code>^ZWfi^M</code>	Write the prestore buffer to a file
<code>^A</code>	Repeat the last search

	Search and replace
<code>^Ax^M</code>	Search for string x
<code>^Ax^Ry^N</code>	Replace all occurrences of string x by string y
<code>^Ax^Ry^M</code>	Replace confirmed occurrences of string x by string y

<code>^A</code>	Ignore this occurrence and repeat the search
<code>^N</code>	Replace all occurrences
<code>^R</code>	Replace this occurrence and repeat the search
<code>^X</code>	Abandon the search

	Spelling checker
<code>^ZS</code>	Disable the spelling checker

<code>^U</code>	Cruise forwards through the WORDS file
<code>^Y</code>	Cruise backwards through the WORDS file
<code>^A</code>	Ignore the word
<code>^A^A</code>	Accept the word and add to the WORDS file
<code>^A^A^A</code>	Disable the spelling checker

Index

The references here are of the form 25-1, indicating the chapter and the page-number within that chapter.

- ! 25-1
- \$INCLUDE 2-12
- \$INSERT 2-12
- %OPEN statement 2-13
- ** operator 2-13, 16-34
- += 2-13
- :ABSLD 22-11
- :FILES 2-20, 21-13
- :SWX 2-4, 4-32
- <Backspace> 8-3
- <Break> 2-2, 6-5, 8-10, 18-1
- <Continuation sequence> 2-4, 2-7
- <Ctrl> key 7-1, 8-3, 8-8, 8-21, App-17
- <Esc> 2-2, 8-12
- <Interrupt> 18-2
- ? 8-20
- @ function 2-13, 16-37, 16-49
- @ function with negative argument 16-49
- @ phrases 2-9, 12-33
- @file.name phrases 12-33
- [in WITH clauses 12-18
- \ operator 2-13, 16-34
-] in WITH clauses 12-18
- ^ in WITH clauses 12-18
- ^ operator 2-13, 16-34
- ^Q 7-3
- ^S 7-3
- ^V 10-12
- ^XC 10-10, 16-88
- ^Xn 10-29
- ^XR 10-10, 16-88
- A processing code 14-1
- A0 2-9, 12-9
- A9998 12-10
- A9999 12-10
- ABS frame 3-5, 4-37
- ABSDUMP 2-21, 21-13
- ACC: See **ACCOUNTS**
- Access 2-6, 2-13, 12-1, 13-1, 14-1
- Access options 12-40
- Access sentence 12-2
- Account 4-15, 4-17, 6-2, 22-2, 22-5
- Account definition 4-16
- ACCOUNT-RESTORE 2-21, 21-14
- Account-save 24-7
- ACCOUNT-SAVE 21-4
- Account synonym 22-7, 23-7
- Accounting statistics 22-16
- ACCOUNTS 4-37, 22-16
- Address 18-15
- Advanced Pick 1-3
- AFTER 12-14
- AP/AT&T Unix 1-4
- AP/DGUX 1-4
- AP/DOS 1-4, 3-1, 25-1
- AP/Native 1-4, 3-1, 25-1
- AP/RS6000 AIX 1-4
- AP/SCO 1-4
- AP/Unix 1-4, 3-1, 19-16, 25-1
- AP: See **Advanced Pick** 1-3
- Apostrophe in heading 12-29
- Application system 6-6
- Argument in subroutines 16-62
- Arithmetic expressions 16-34
- Arithmetic operators 16-34
- Arrays: See **Dimensioned arrays**;
Dynamic arrays
- ASCII characters 16-38, App-15
- Assembly language 2-27
- ASSIGNED 2-13, 16-36
- ASSIGNFQ 2-22, 19-7, 19-16
- Assignment statements 16-34
- Attribute 4-2
- Attribute-count 13-2
- Attribute defining item 13-1

- Attribute definition 5-7, 10-5, 10-7, 13-1, 13-9
Attribute-mark 4-2, App-16
Attribute-type 13-2, 13-7
- B 10-15, 14-6
B-tree 2-16, 15-5
B/ADD 2-4
B/LIST 2-12, 16-33
Backing storage 2-20, 3-2, 3-11, 16-83, 20-1, 25-3
Backup 3-11
Backward link 4-6, 4-20
Base FID 4-18, 4-29
Base frame 4-5
Basic 2-11, 16-1
BASIC 16-86, 16-92
Basic debugger 6-5, 16-87, 18-1, 18-4
Basic functions 2-13, 16-37, App-13
BASIC options 16-87, 16-92
Basic statements 2-13, App-12
Batch processing 16-77
BEFORE 12-14
Blank lines in programs 2-12, 16-5
BLIST 2-12
BLOCK-CONVERT 4-37
BP 4-37
BREAK-ON 12-21, 12-25, App-10
BREAK-ON options 12-23
Bridges 10-15, 14-6
BRK-DEBUG 8-12
BRK-LEVEL 8-12
BSYM 16-87
Buffer 17-8
BULLETIN 22-16
BULLETIN.BOARD 22-2, 22-16
BY 2-6, 12-19
BY-DSND 12-20
BY-EXP 12-21
BY-EXP-DSND 12-21
- C 14-7
CALL 14-8, 16-60
CAPTCL 2-5, 8-4, 8-10, 23-8
CAPTURE 8-10
CAPTURING 16-67
- Capturing TCL history 8-10, 23-8
Cartridge tape 3-11, 20-1
CASE 8-5, 16-28
Case sensitivity 2-2, 2-12, 4-28, 8-3, 8-5, 16-58
CASING 2-13
CATALOG 16-90
Catalogue-pointer 2-12, 16-90
CHAIN 16-69
CHAR 16-38
CHARGE-TO 22-17
CHARGES 22-17
CHECK-ACCOUNT 22-12, 24-5
CHECK-DX 21-7
CHECK-FILES 22-12, 24-5
CL 2-11, 12-36
CLASHES 2-2, 2-4, 2-26
CLEAR 16-36
CLEAR-FILE 2-5, 4-26
CLEARFILE 16-77
CLOSE 2-14
Closed system 22-4, 23-6
COL-HDR-SUPP 12-30, 12-40
Colon in PRINT statements 16-46
Column-width 13-2, 13-8, 13-10
Comma in PRINT statements 16-46
COMMENT 8-16,
COMMON 2-14, 16-62, 16-65
COMPARE 4-50, 12-40
COMPARE-LIST 12-38, 12-40
Comparison operators 12-14
COMPILE 2-12, 16-86, 16-92
COMPILE-CATALOG 2-12, 16-88
COMPILE options 16-87, 16-92
COMPILE-RUN 2-12, 16-88
Computed GOSUB 16-30
Computed GOTO 16-30
Concatenation 14-7
Connective 12-30
Control break 12-21
Control field 4-20
Controlling attribute 13-4
CONVERSE 2-5
Conversion 2-11, 10-30, 13-8, 13-11, 14-1
CONVERT 2-14

- Converting from R83 to AP 2-23
- COPY 4-46
- COPY-LIST 2-11, 12-36
- Correlative 2-11, 10-30, 13-2, 13-6, 13-8, 13-11, 14-1
- COUNT 12-4, 16-38
- CP 4-44
- CPU 3-2
- CREATE-ACCOUNT 22-5, 22-6
- CREATE-BFILE 2-11
- CREATE-FILE 4-10, 4-14, 4-22
- CREATE-INDEX 14-15, 15-2
- CREATE-MACRO 8-15
- CRT 16-46
- Cruising 10-15, 10-17, 10-19, 14-15, 15-1
- CT 4-44
- CU 14-10
- CURSOR 2-28
- Cursor control keys 7-1
- Cursor positioning 16-49
- Cut and paste 10-25, App-18
- Cut-id 10-26

- D 14-10
- D-pointer 2-19, 4-12, 4-17, 4-23, 4-28, 4-30, 10-5
- DAT tape 20-1
- DATA 2-5, 16-55, 16-67
- Data design 5-9
- DATA-ENTRY 10-4
- Data-level identifier 4-12
- Data model 5-1
- Data normalisation 5-1
- Data section 4-10, 4-13
- Data transparency 4-36
- Dataname 4-10, 5-7, 12-2, 13-1, 14-3, 16-80
- Date 10-11, 14-10, 14-12, 22-13
- DATE 16-38
- DBL-SPC 12-30
- DC-pointer 2-19
- DCOUNT 16-38
- DEBUG 18-6
- Debugger 6-5, 18-1
- DECATALOG 16-91
- Default output list 2-9, 12-12, 12-33
- DEFINE-TERMINAL 2-5, 2-28, 4-37, 22-18
- DEL 2-14, 16-42
- DELETE 4-45, 16-43, 16-77
- DELETE-ACCOUNT 22-6
- DELETE-FILE 2-5, 4-27
- DELETE-INDEX 15-4
- DELETE-LIST 2-11, 12-36
- Deleting items 4-22, 4-45
- Dependent attribute 13-4
- Description 4-29
- Despooling 19-2
- DET-SUPP 12-24, 12-40
- DEVICE 22-18
- DEVICES 2-22, 2-28, 4-37, 22-18
- DFSIZE 22-8
- DICT 4-10
- DICT-only file 4-12
- Dictionary 4-10, 12-2
- Dictionary-code: 13-2
- Dictionary design 5-6
- Differences 2-1
- DIM 16-12
- DIMENSION 2-14, 16-12
- Dimensioned arrays 2-14, 2-16, 16-12, 16-75
- Direct access 5-3
- Direct item 4-20, 5-4
- Dirty bits 2-19, 2-21, 4-21, 21-10
- Disk storage 22-9
- Diskette 20-1
- Displacement 18-15
- DISPLAY 8-16
- DL 2-11, 12-36
- DL-pointer 21-19
- DM 2-18, 4-16
- Double-clutching 10-17, 10-19
- DS-pointer 8-7
- Dummy-restore 21-11
- DUMP 18-15
- DUPLICATE 12-30, 15-4
- DX-pointer 21-6
- DY-pointer 21-6
- Dynamic arrays 2-16, 16-16, 16-42, 16-75
- EACH 12-16

- ED 9-3
- EDIT 9-3
- EDIT-LIST 2-11, 12-36
- Editor 9-1
- Editor commands 9-4
- EL 2-11, 12-36
- Emulation 3-3
- End-of-file marker 20-2
- End-of-group marker 4-20
- End-of-item marker 4-3, 4-20
- ENVIRON 25-2
- EOF 20-2
- EPick 2-5, 8-3, 8-20
- EQUAL 12-14
- EQUAL 2-7
- ERRMSG 2-18
- ERROR 2-15, 16-70
- ERROR() 2-15
- ERRORS 2-18, 4-37, 22-12, 24-5
- ERRORS,GFE 24-6
- ESC-DATA 8-12
- ESC-LEVEL 8-12
- ESC-TOGGLE 8-12
- EVERY 12-16
- EXCHANGE 4-43
- EXECUTE 2-15, 16-66
- EXIT 2-15, 2-16, 25-1
- EXPORT 25-2, 25-3
- External subroutine 16-59, 16-90
- EXTRACT 16-42

- F 14-13
- F-RESIZE 21-20
- FID 3-4; **See Frame identifier**
- FIELD 16-38
- Field 4-1; **See Attribute**
- File 4-1
- FILE 2-15, 16-80
- File defining item 4-12, 4-16
- File definition 4-12, 4-17
- File design 5-1
- File efficiency 5-2
- File handling (Basic) 16-72
- File handling (Procs) 17-18
- File indexing 14-15, 15-1
- File name 5-2
- FILE-OF-FILES 2-18, 2-21, 4-27, 4-37, 21-9, 21-20
- File resizing 21-19
- File-restore 2-20, 21-12
- File-save 2-20, 24-7
- FILE-SAVE 21-3
- File statistics 21-9
- File structure 4-10
- File synonym 4-15, 4-17, 4-33
- File-variable 2-17, 16-72
- FILL 2-7, 12-8, 12-40
- FIND 4-51
- FIND2 4-51
- Fixed disk 3-2
- FL 12-37
- Flash/Basic 2-12, 16-87, 16-91
- Floppy diskette 3-11, 20-1
- FOLD 2-15, 16-38
- FOOTING 2-15, 12-28, 16-51
- FOOTING options 2-7, 12-29, App-10
- FOR / NEXT 16-30
- Form-queue 19-3, 19-9
- FORMAT 20-6.
- Format mask 16-47
- Forward link 4-6, 4-20
- Frame 3-4
- Frame fault 3-10, 24-5
- Frame identifier 3-4
- Frame size 2-2
- FREE 22-10
- FS 14-14
- FUNCKEYS 4-37
- Function keys 7-1

- G 14-14
- GET 2-15
- GET-LIST 2-11, 12-34
- GETX 2-15
- GFE 18-19, 21-10, 22-11, 24-1
- GFE encountered 24-6
- GFE-FID 24-6
- GFE-FILE 24-6
- GIRLS 1-1
- GL 2-11
- Global dictionary 12-11
- Global variable 16-62
- Glossary App-1

- GOSUB 16-59
- GRAND-TOTAL 12-25
- GRAND-TOTAL options 12-25
- GraPICK 8-23, 10-2
- Group 4-7, 4-25, 24-1
- Group extraction 14-14
- Group format error: See **GFE**
- Hard disk 3-2, 3-4
- HASH-TEST 4-25, 12-4
- Hashing 4-7
- Hashing algorithm 4-8
- HDR-SUPP 12-30, 12-40
- Header label 20-2
- HEADING 2-15, 12-28, 16-51
- HEADING options 2-7, 12-29, App-10
- Help 10-13
- HELP 2-5, 8-3, 8-20
- Hierarchy of files 4-16
- Host operating system 25-1
- Hotkey 4-29, 10-29
- Hotkey.all 10-29
- I 14-15
- ICONV 2-11, 10-30, 14-1, 16-38, 16-40
- ID 10-4, 14-16
- ID-PROMPT 10-4
- ID-SUPP 12-30, 12-40
- IF (Access) 12-15, 14-18
- IF (Basic) 16-24
- IF (Procs) 17-15
- IFNO 2-7, 12-16
- Implicit output list 2-9, 12-12
- IMPORT 25-3
- IN 2-15, 16-55
- In-line item 4-22; See **Direct item**
- INCLUDE 2-12, 16-66, 16-87
- Incremental file-save 2-21, 4-22, 21-3, 21-8
- Indentation 16-7, 16-33
- Index 2-16, 2-19, 10-16, 14-15, 15-1
- INDEX 16-38
- Indirect item 4-21, 4-28, 5-4
- INPUT 2-15, 16-54
- INPUT @ 16-56
- Input buffer 2-4
- Input-conversion 4-29, 10-30, 14-1
- INPUTERR 16-57
- INPUTNULL 16-57
- INPUTTRAP 16-57
- INS 2-14, 16-42
- INSERT 2-12, 16-43
- INT 16-39
- Internal subroutine 16-59
- Intrinsic functions 16-37
- IOMAP-FILE 4-37
- IS 2-7, 12-14
- ISELECT 12-37
- ISSELECT 12-37
- ISTAT 4-25, 12-4
- Item 4-1, 4-19, 4-25
- Item format 4-19
- Item-id 2-5, 2-6, 4-1, 4-20, 10-3, 12-13, 14-16
- Item-id design 5-8
- Item-list 2-4, 8-4
- Item lock 2-13
- Item size 2-19
- Items listed 2-8; See **NI-SUPP**
- Iteration 16-30
- JOBS 4-37
- Julian date 14-12
- KB.FK 4-37
- KB.PC 4-37
- Key 4-1
- KEY 2-16, 15-5
- Keyboard 7-1
- KEYBOARDS 4-37, 7-4
- KEYS 22-3, 22-5, 23-5
- L 14-19
- L/RET: See **RET-LOCK**
- L/UPD: See **UPD-LOCK**
- Label 20-2
- Labelled COMMON 2-14, 16-65
- Labels 12-41
- Leaf 15-8
- LEG-SUPP 2-6, 12-28, 12-40
- LEGEND 2-7, 4-40, 12-27

- LEN 16-39
- Length 14-19
- LET 2-16
- Level pushing 2-2, 8-10
- Level pushing and LOGTO 8-13
- Limit: form-queues 19-4, 19-6
- Limit: length of item-id 5-8
- Limit: length of strings 2-12, 16-9
- Limit: level pushing 8-13
- Limit: number of variables 2-12, 16-10
- Limit: numeric values 16-11
- Limit: size of items 2-19
- Limit: spooler jobs 19-4
- Limit: variable names 16-10
- Line 3-3
- LIST 12-4
- LIST-DEVICE 20-5
- LIST-ITEM 4-44, 12-4
- LIST-LABEL 12-4, 12-41
- LIST-LISTS 2-11, 12-37
- LIST-LOCKS 22-9
- LIST-MACROS 8-15
- LIST-MENU 8-19
- LIST-MENUS 8-19
- LIST-OBJ 16-89
- LISTABS 19-7, 19-8
- LISTACC 4-38, 22-16
- LISTCONN 12-32
- LISTDICT 12-8
- LISTFILES 4-23, 12-6, 22-6
- LISTPEQS 19-11
- LISTPTR 19-7
- LISTU 22-16, 23-8
- LISTUSERS 22-4
- Literal 16-8
- LL 2-11, 12-37
- Local variable 16-61
- LOCATE 16-44
- Lock 2-13, 2-19
- Lock-code 4-18, 4-29, 22-3, 23-3
- Lock table 22-9
- LOCKED 16-80
- Locking 16-79
- LOG-MSG 23-8
- LOG-STATUS 21-19
- Logging off 6-5
- Logging on 6-2, 23-1
- Logical connective 16-35
- Logical expressions 16-25
- Logical operators 16-35
- LOGON 4-40, 22-15
- Logon Proc 2-2
- Logon procedure 2-2, 22-2, 23-5
- LOGTO 2-5, 22-2
- LOGTO and level pushing 8-13
- LOOP 16-31
- LPTR 12-26, 12-40
- Machine code 16-4
- Macro 2-2, 2-9, 2-28, 4-29, 10-3, 12-12
- MACRO 22-2, 22-4
- Macro definition 8-14
- Macros 8-13
- Macros in OP 11-6
- Magnetic tape 3-11, 20-1
- Mask 16-47
- Mask character 14-19
- Master dictionary: See MD
- MAT 16-15
- MATBUILD 2-16, 16-21, 16-44
- MATPARSE 2-16, 16-21, 16-44
- MATREAD 16-74
- MATREADU 16-80
- MATWRITE 16-74
- MATWRITEU 16-80
- MAXFID 22-8
- MC 14-19
- MC/AN 14-20
- MC/N 14-20
- MC/NA 14-20
- MCA 14-20
- MCAN 14-20
- McDonnell Douglas 1-1
- MCDX 14-20
- MCL 14-20
- MCN 14-20
- MCNA 14-20
- MCP 14-20
- MCS 14-20
- MCT 14-20
- MCU 14-21
- MCXD 14-20

- MD 4-17
- MD (processing code) 14-21
- MDS 2-18, 4-15, 4-17
- Memory 3-2, 22-8
- Menu 8-17
- Menu definition 8-17
- Message 22-14
- MESSAGE 2-5
- MESSAGES 2-18, 4-37, 4-39, 4-41, 12-27
- MI 14-28
- Microdata 1-1
- ML 14-21
- MOD 16-39
- Modifier 12-30
- Modulo 2-19, 4-7, 4-11, 4-18, 4-29, 5-2, 5-4, 21-20
- MOVE-FILE 4-32
- Moving to Advanced Pick 2-1
- MP 14-28
- MR 14-21
- MS 14-28
- MSG 2-5, 22-14
- MT 14-22
- Multiple account-save 2-21, 21-4
- Multuser interface 2-20, 20-1
- Multivalued 4-3, 10-13, 12-16
- MX 14-29
- MY 14-29

- Named COMMON 16-65
- Nelson, Don 1-1
- NEWAC 4-37
- NEXT 16-30
- NFRAME-INDEX 15-4
- NI-SUPP 2-8, 12-30, 12-40
- Non-columnar output 12-8
- NOPAGE 12-30, 12-40
- Normalisation 5-1
- NOT 12-14, 16-39
- NSELECT 12-37
- NUM 16-39
- Number 16-8
- Numeric dataname 12-12, 12-33
- Numeric datanames 2-9
- Numeric keypad 7-1
- Numeric string 16-11

- O 14-29
- OA: See **Open Architecture** 1-3
- Object program 16-4, 16-88
- OCCURS 2-16
- OCONV 2-11, 10-30, 14-1, 16-38
- ON ... GOSUB 16-30
- ON ... GOTO 16-30
- ONERR 2-16, 16-83
- ONLY 2-9, 10-3, 12-13
- OP 11-3, 11-5
- OP commands 11-5, App-8
- OP macros 11-6
- OP options 11-5
- OPEN 16-72
- Open Architecture 1-3
- Operations 21-1, 22-1
- Operator assignment 2-13, 16-36
- OPTIONS 22-4, 23-7
- OUT 2-15, 16-56
- Out-of-line item 4-21; See **Indirect item**
- Output-conversion 13-2, 13-6, 14-1
- Output list 2-8, 12-6
- Output-macro 4-29, 12-12
- Output Processor 2-22, 11-11 See **OP**
- Overflow 3-6, 4-4
- OVERFLOW 2-5, 22-8, 22-10
- Overflow reserve 3-7
- OVF: See **OVERFLOW**

- P 14-23
- P option 12-27
- PA 2-18
- Packed decimal 14-28
- PAGE 16-52
- Page number 12-29
- Parallel printer 3-3
- Parameter 16-62
- Partial string 12-17
- Password 6-2, 23-1
- PASSWORD 22-3, 22-5
- Paste 10-25, App-18
- Pathname 2-20, 4-36, 12-6
- Pattern 14-23
- Pattern matching 14-23, 16-27

- PCB 3-5, 3-6, 22-8
 Phantom process 8-22
 Phrase 2-9, 12-12, 12-32
 PIB 3-3, 22-8
 PIBS 2-28, 4-37, 22-16, 22-18, 22-20
 Pick & Associates 1-1
 Pick Systems 1-2
 Pick, Dick 1-1
 PK Harmony 3-3
 POINTER-FILE 2-9, 2-18, 4-37, 12-38
 Polish notation 14-13
 Port 3-2, 3-3
 Postfix notation 14-13
 POVf: See **OVERFLOW**
 PQ 17-8
 PRECISION 2-16, 16-11
 Predefined phrases 2-9
 Prestore commands 9-11, 10-23, App-19
 Prestore-id 10-24
 Primary control block 3-5
 Primary file space 4-5
 Primary select-list 2-10, 12-39
 PRIME 5-6
 PRINT 16-46
 PRINT-ERR 4-41
 PRINT ON 16-51, 19-10
 Printer 3-2, 6-8
 PRINTER ON 16-50
 Printer width 2-6
 Printing 19-1
 PRIVILEGE 22-4
 Proc statements App-14
 Processing code 2-2, 2-11, 2-27, 10-30, 14-1, 16-40
 PROCLIB 2-18, 2-27
 PROCREAD 16-71
 Procs 2-26, 17-1, App-14
 PROCWRITE 16-71
 Program indentation 16-7, 16-33
 Program structure 16-23
 PROMPT 8-16, 16-54
 Pseudo-object code 16-4
 PSH 25-2
 Q-pointer 2-20, 4-17, 4-33, 10-5
 QA 2-18
 QFILE 4-35
 QSELECT 12-36
 Quantum 3-10
 R 14-24
 R83 Pick 1-2
 R83.SETUP 2-3
 Random number 16-39
 Range checking 14-24
 READ 2-17, 16-73
 READNEXT 2-17, 12-40, 16-78
 READT 16-83
 READTL 2-17, 16-84
 READTX 2-17, 16-84
 READU 16-80
 READV 16-74
 READVU 16-80
 Reallocation 4-29, 21-20, 22-6
 REALLOCATION 22-6
 Record 4-1; See **Item**
 Record key 4-1; See **Item-id**
 RECOVER-FD 4-46, 9-14
 Redefining the keyboard 7-3
 REFORMAT 2-8, 12-4, 12-43
 REM 16-39
 Remote phrases 12-32
 RENAME-FILE 4-33
 Repeating group 5-1
 Replace 10-27, App-19
 REPLACE 2-17, 16-43
 Reserved word 16-10
 RESIZE 21-21
 Resizing 21-19
 RESTORE-ACCOUNTS 2-21, 21-15
 RET-LOCK 4-29, 22-5, 23-5
 RETURN 16-59
 RETURNING 16-68
 Reverse Polish notation 14-13
 REWIND 16-84
 RND 16-39
 ROLL-ON 2-8, 12-23, App-10
 ROOT 2-17, 15-4, 15-8
 RQM 3-11
 RUN 16-89

- RUN-LIST 2-12, 16-90
RUN options 16-89
Runoff 2-22, 11-8, App-9
Runoff commands 11-8, 11-11, App-9
Runoff control characters 11-11
RUNOFF options 11-10
- S 14-24
S-DUMP 12-5, 20-9
SAMPLING 2-8, 12-13
SAVE 2-21, 21-10, 22-12, 24-5
SAVE-LIST 2-9, 2-11, 4-37, 12-34
Saved-list 2-9, 12-34
Scaling factor in masks 16-47
Scrub with CLEAR-FILE 4-27
Search and replace 10-27, App-19
SEARCH-FILE 4-51
SEARCH-SYSTEM 4-51
SECONDARY 2-10, 2-17, 12-40, 16-79
Secondary select-list 2-10, 2-17, 4-49, 12-39, 16-79
Sector 3-4
Security 22-3, 23-1
Segment-mark 4-3, App-16
SEL-RESTORE 21-12, 21-15, 21-18
SELECT 2-17, 12-5, 12-34, 16-78
Select-list 2-9, 12-34
Select-variable 16-78
Selection criterion 12-14
Selective restore: See **SEL-RESTORE**
SEND 2-15
SEND-MESSAGE 2-5, 22-14
SENDX 2-15
Separation 2-19
SEQ 4-41, 14-29, 16-39
Sequential processing 5-3
Serial printer 3-3
SET-8MM 20-3
SET-BAUD 22-20
SET-DATE 22-13
SET-DATE-EUR 22-13
SET-DATE-STD 22-13
SET-DEVICE 2-20, 20-3, 20-4
SET-DPTR 2-4, 2-19, 4-23, 4-31, 21-7, 21-19
SET-FILE 4-35
SET-FLOPPY 2-20, 20-3
SET-FUNC 7-6
SET-HALF 20-3
SET-KBRD 7-5
SET-KEYS 7-5
SET-OVF-RESERVE 3-7, 22-11
SET-PORT 22-20
SET-SCT 20-3, 20-4
SET-SCT-DMA 20-3
SET-TAPE-TYPE 20-3
SET-TERM 2-28, 22-20
SET-TIME 22-14
Shelling out 25-1
Single-user interface 20-1
SL 2-11
Slave printer 3-3
SLEEP 3-11, 8-21
Soft GFE 24-4
SORT 12-5, 16-44
SORT function 2-17
SORT-ITEM 4-44, 12-4
SORT-LABEL 12-4, 12-41
SORT-LIST 12-37
Sort specification 12-19
SOUNDEX 2-17
Source program 16-3
SP-ASSIGN 16-51, 19-8
SP-EDIT 19-12
SP-KILL 19-10
SP-STATUS 19-7, 22-9
SP-TAPEOUT 19-13
SPACE 16-39
SPELLER 10-28
Spelling checker 10-28, App-19
Spooler 2-22, 3-2, 16-50, 19-1
Spooler assignments 19-8
Spooler status report 22-9
Spreadsheet 2-8, 12-31
SQL 12-45
SQRT 16-39
SREFORMAT 2-8, 12-4, 12-43
SS 2-8, 12-31
SSELECT 12-5, 12-35
Stack (Procs) 17-8

- STACK-ON 8-9
- Stacker 2-5, 4-38, 6-3, 8-4, 8-8, 23-8, App-17
- Stamp 14-28
- Standard file 4-37
- START.SS 19-5
- Starting the system 22-1
- STARTLOG 21-19
- STARTPTR 19-6
- STARTSPOOLER 2-22, 19-5
- STAT 12-5
- STAT-FILE 2-18, 2-21, 4-40; See **FILE-OF-FILES**
- Statement label 16-5, 16-23
- Status report 22-9
- STEAL-FILE 4-33
- STEP 16-31
- STOFF 17-10
- STON 17-10
- STOPLOG 21-19
- Stopping the system 22-1
- STOPPTR 19-8
- STR 16-39
- Streamer cartridge tape 3-11, 20-1
- String 16-8
- String expressions 16-34
- String operators 16-34
- Structure: 13-2, 13-4
- Subroutine (Basic) 16-58
- Subroutine (Procs) 17-16
- Subroutine arguments 16-62
- Subscript 16-13
- Substitute-header 13-2, 13-3, 13-8, 13-10
- Substring 16-34
- Substring assignment 2-13
- Substring replacement 16-34
- Subtotal 12-24, 12-25
- Subvalue 4-3, 10-14, 16-76
- Subvalue-mark 4-3, App-16
- SUM 12-5, 16-44
- SUM function 2-17
- SUPP 12-40
- Switching off 6-6
- Switching on 6-1
- Symbol table 16-87, 18-7
- Synonym 4-15
- SYSDBASE 3-6, 22-8
- SYSPRIV 22-6, 23-3
- SYSPROG: See **DM**
- SYSTEM 2-18, 16-39, 16-41
- System administration 22-1
- System architecture 3-1
- SYSTEM-COLDSTART 22-1
- System debugger 6-5, 18-1, 18-13
- SYSTEM-ERRORS 2-18
- SYSTEM function 2-18
- System privileges 23-2
- SYSTEM(0) 2-16
- T 14-25
- T-ACT 2-20, 20-6
- T-ATT 20-5, 20-6
- T-BCK 20-6
- T-BSF 20-6
- T-BSR 20-6
- T-CHK 20-6, 21-12
- T-DET 2-20, 20-5, 20-6
- T-DUMP 2-20, 12-5, 12-40, 20-9
- T-EOD 20-7
- T-ERASE 20-7
- T-FSF 20-7
- T-FSR 20-7
- T-FWD 20-7
- T-LINK 20-7
- T-LOAD 2-20, 12-5, 20-9
- T-RDLBL 20-7
- T-READ 20-7
- T-RET 20-7
- T-RETEN 20-7
- T-REW 20-7
- T-SELECT 20-7
- T-SPACE 20-8
- T-STAT 20-8
- T-STATUS 20-8
- T-UNLD 20-8
- T-UNLOAD 20-8
- T-VERIFY 20-8, 21-11
- T-WEOF 20-8
- T-WTLBL 20-8
- TANDEM 2-6
- Tape 20-1, 20-10
- TAPE 21-12
- TCL 2-2, 6-3, 8-1, 8-3, 16-69

- TCL buffer 2-4, 2-7
- TCL capturing 2-5
- TCL commands 2-18
- TCL-HDR 2-8, 12-27
- TCL history 8-10
- TCL options 8-4, 12-27
- TCL prompt 2-2, 8-1
- TCL-STACK 4-38, 8-9, 23-8
- TCL stacker 2-5, 6-3
- TCL-SUPP 2-8, 12-27
- TCL verb 8-2
- TCLREAD 2-18, 16-70
- Temporary attribute item 2-9, 12-9
- TERM 2-6, 22-18
- TERM-TYPE 2-28, 22-2, 22-18
- Terminal 3-2, 6-1
- Terminal characteristics 22-17
- Terminal command language: See **TCL**
- Terminal emulation 3-3
- Terminal name 22-18
- Termite 3-3
- Text processing 11-1
- Text substring 14-25
- Tfile 14-25
- Throwaway modifier 12-31
- Time 10-12, 14-22, 22-13
- TIME 16-39
- Time-out 2-15, 16-54
- Time-slice 3-10
- TIMEDATE 2-3, 8-3, 8-7, 16-39
- TLOG-RESTORE 21-19
- TO 2-5; See **LOGTO**
- TOTAL 12-24
- Trace table 18-7, 18-8
- Track 3-4
- Transaction logging 2-21, 4-28, 21-18
- TRIM 16-40
- TUTOR 2-18
- TXLOG 21-19
- TXLOG-STATUS 21-19
- TYPE 8-16

- U 10-1
- U01AD 17-18
- U50BB 2-6
- U60BB 2-6
- UD 4-23, 4-30, 10-5
- Unix 1-2, 1-4
- UP 10-1
- UPD-LOCK 4-29, 22-5, 23-5
- UPDATE 4-43, 10-1
- UPDATE-ACCOUNT 2-4, 2-25
- UPDATE-LOGGING 21-19
- UPDATE-MD 2-25
- UPDATE options 10-2
- Update Processor 10-1
- Update Processor vs application programs 10-8
- Update stamp 14-28
- User 22-2
- User definition 2-1, 2-2, 2-28
- User-exit 2-2, 2-27, 14-29
- User-id 2-1, 2-28, 6-2, 22-3, 23-1
- User status report 22-9
- USERS 2-28, 22-2, 23-2, 24-6
- USING 12-10

- V 14-29
- Value 4-3, 16-76
- Value-mark 4-3, App-16
- Variables 16-8, 16-10
- Variable names 2-12, 16-10
- Verb 8-2, 12-3, 12-4
- VERIFY-ABS 22-11
- VERIFY-INDEX 15-4
- VERIFY-SYSTEM 22-11
- Verifying the save 21-11
- VERSION 4-41
- Viaduct 3-3
- Virtual memory 3-2, 3-4, 3-8
- Volume 20-1

- WEOF 16-84
- WHAT 22-8
- WHERE 22-9, 23-8
- WHO 2-6, 23-8
- Wild card in array subscripts 2-15, 16-16
- Wild card in WITH clauses 12-18
- Window 18-15, 18-16
- WITH 12-14
- WITH EACH 12-16

WITH EVERY 12-16
WITHOUT 2-7, 12-16
WLIST / WSORT 15-4
WORDS App-19
Work space 3-5
WRITE 16-74, 16-76
WRITET 16-84
WRITETX 2-17
WRITEU 16-80
WRITEV 16-74, 16-76
WRITEVU 16-80
WSELECT / WSSELECT 15-4
WSYM 11-5

X 14-29
Xc 14-28
XC 13-8
XI 14-29
XO 14-29
XR 12-25, 14-29
XT 14-29

Z 8-22
ZA 14-30
ZC 14-30
ZHS 8-22
Zooming 10-17, 10-19

✦

✦