

# Paradox<sup>®</sup>

Version 4.5

---

## PAL<sup>™</sup> Programmer's Guide

Borland International 1800 Green Hills Road  
P.O. Box 660001, Scotts Valley, CA 95067-0001, USA

Borland may have patents and/or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents.

COPYRIGHT 1985, 1993 by Borland International. All rights reserved. All Borland products are trademarks or registered trademarks of Borland International, Inc. Paradox is a registered trademark of Borland International, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Borland International, Inc. 1800 Green Hills Road, P.O. BOX 660001, Scotts Valley, CA 95067-0001

PRINTED IN IRELAND

10 9 8 7 6 5 4 3 2 1

R1



# CONTENTS

<b>Chapter 1</b>			
<b>Introduction</b>	1		
What is PAL?	2		
PAL and Paradox	2		
PAL facilities	5		
Why use PAL?	6		
Before you use PAL	6		
How to use this manual	7		
Printing conventions	8		
Standard and compatible modes	9		
Preparing existing applications for compatible mode	9		
New commands and functions in Paradox 4.0	11		
Commands and functions prior to Paradox 4.0	11		
<b>Part I</b>			
<b>Basic building blocks</b>	13		
<b>Chapter 2</b>			
<b>Scripts and commands</b>	15		
The nature of a script	15		
Commands	17		
Syntax	17		
Case	18		
Comments and blank lines	18		
Sequence of execution	19		
Scripts within scripts	20		
Types of PAL commands	20		
Programming commands and functions	21		
Keypress interactions	22		
Menu choices	22		
Special keys	23		
Quoted strings	26		
Query images	26		
		Abbreviated menu commands	27
<b>Chapter 3</b>			
<b>Expressions</b>	31		
What is an expression?	31		
Data types	32		
Evaluating expressions	33		
Elements of expressions	34		
Constants	34		
Alphanumeric constants	34		
Special characters	34		
Numeric constants and the international number format	35		
Date constants	36		
Logical constants	37		
Blank values	38		
Variables	38		
Naming conventions	38		
Assigning values to variables	40		
Using variables	40		
System variables	42		
Using Retval	42		
Fixed and dynamic arrays	43		
Fixed arrays	43		
Fixed array elements	43		
Lifetime of arrays and array elements	44		
Using arrays with records	45		
Dynamic arrays	45		
Declaring dynamic arrays	45		
Dynamic array elements	45		
Operators	46		
The + operator	47		
The - operator	47		
The * and / operators	48		
Comparison operators	48		
Logical operators (AND, OR, NOT)	49		
Order of evaluation	50		
Field specifiers	50		
Using field specifiers in expressions	51		

Moving to specific fields . . . . .	53	Optional elements . . . . .	81
Functions . . . . .	53	Alternative choices . . . . .	82
Manipulating values . . . . .	54	Inhibiting automatic fill-in . . . . .	84
String manipulation . . . . .	54	Picture examples . . . . .	84
Date manipulation . . . . .	55	Using format specifications to control	
Date formats . . . . .	55	output . . . . .	85
Date operations . . . . .	56	Width . . . . .	86
Date functions . . . . .	57	Alignment . . . . .	87
Date patterns . . . . .	57	Case . . . . .	87
PAL keycodes . . . . .	58	Edit . . . . .	87
Single-character strings . . . . .	58	Sign . . . . .	88
Function keys . . . . .	59	Date . . . . .	89
Special Paradox key names . . . . .	59	Logical . . . . .	89
Positive numbers representing ASCII		<b>Chapter 6</b>	
codes . . . . .	59	<b>Procedures</b>	91
Negative numbers representing IBM		What is a procedure? . . . . .	91
extended codes . . . . .	60	Defining procedures . . . . .	92
Using keycodes in expressions . . . . .	60	Calling procedures . . . . .	94
Converting codes . . . . .	61	Input to procedures: Parameters and	
<b>Chapter 4</b>		arguments . . . . .	94
<b>Control structures</b>	63	Output from procedures: Return values . . . . .	97
Conditions . . . . .	63	A note on Retval . . . . .	98
Branching . . . . .	64	Variables and procedures . . . . .	99
IF . . . . .	65	Dynamic scoping . . . . .	100
SWITCH . . . . .	66	Formal parameter variables . . . . .	101
Branching with dynamic arrays . . . . .	68	Single variable arguments . . . . .	101
IIF ( ) . . . . .	69	Array arguments . . . . .	102
Loops . . . . .	69	Variables declared private . . . . .	102
FOR . . . . .	70	Closed procedures . . . . .	105
FOREACH . . . . .	71	Procedures and libraries . . . . .	107
WHILE . . . . .	72	Creating procedure libraries . . . . .	108
SCAN . . . . .	73	Storing procedures in libraries . . . . .	109
Interrupting a loop . . . . .	74	Using procedures from libraries . . . . .	109
Returning and exiting . . . . .	75	Autoloading procedures . . . . .	109
RETURN . . . . .	75	Explicitly loading procedures . . . . .	110
QUIT and EXIT . . . . .	76	Listing the contents of a library . . . . .	112
<b>Chapter 5</b>		More on WRITELIB . . . . .	113
<b>Formatting data</b>	77	Using libraries from previous versions of	
Using pictures to format input . . . . .	77	Paradox . . . . .	113
How to define pictures . . . . .	78	Debugging procedures stored in libraries . . . . .	114
Special features of pictures . . . . .	80		
Repetition counts . . . . .	80		

<b>Chapter 7</b>			
<b>Special topics</b>	115	Instant Script Record	140
Password protection	115	Scripts   BeginRecord	141
Special scripts	119	PAL Menu   BeginRecord	141
Init	120	Scripts   QuerySave	141
Instant	120	What recorded scripts look like	143
Errors	120	What is recorded	144
Types of errors	120	Editors	144
Error processing	121	The Paradox Editor	144
Error procedures	123	ASCII text editors	145
Running external programs from		Summary	146
Paradox	125	Playing scripts	146
		Scripts   Play	147
		Play from the PAL menu	147
		Scripts   RepeatPlay and RepeatPlay from	
		the PAL menu	147
		Instant Script Play	148
		Playing scripts from DOS	148
		Script errors and interruptions	148
		Interrupting script play	149
<b>Part II</b>			
<b>PAL facilities</b>	127		
<b>Chapter 8</b>			
<b>The PAL menu</b>	129		
Displaying the PAL menu	129	<b>Chapter 10</b>	
Play	130	<b>The PAL Debugger</b>	151
RepeatPlay	130	Starting the Debugger	152
BeginRecord	130	Working with the Debugger	153
Debug	131	Debugging levels	153
Value	131	Debugging procedures from libraries	154
Examining the values of variables and		The Debugger screen	154
array elements	132	Debugging keys	155
Errors in expressions	132	The Debugger menu	156
MiniScript	133	Value	156
Command-line access to PAL and Paradox	134	Step	157
Using miniscripts in testing	134	Trace	158
Using miniscripts to set up keyboard		Next	158
macros	134	Go	158
Using miniscripts to learn PAL	135	MiniScript	159
Screen display by miniscripts	135	Where?	160
Miniscript errors	136	Quit	160
Quitting the PAL menu	137	Pop	160
		Editor	161
		Quitting the Debugger	161
		A Debugger tutorial	162
<b>Chapter 9</b>			
<b>Creating and playing scripts</b>	139		
Creating scripts	139		
Recording keystrokes	140		

<b>Part III</b>	
<b>Interacting with the user</b>	169
<b>Chapter 11</b>	
<b>Using windows</b>	171
Where the action takes place . . . . .	171
Objects in windows . . . . .	171
Creating a window . . . . .	172
Getting a window handle . . . . .	172
Testing a handle . . . . .	174
Working with window attributes . . . . .	174
Getting window attributes . . . . .	174
Setting window attributes . . . . .	176
Sizing and shaping windows . . . . .	177
Specifying the current window . . . . .	178
Closing windows . . . . .	179
The three layers of Paradox . . . . .	179
The desktop layer . . . . .	181
The echo layer . . . . .	181
The application layer . . . . .	182
The workspace and the desktop . . . . .	182
Floating windows . . . . .	182
Floating windows and desktop objects . . . . .	183
<b>Chapter 12</b>	
<b>Controlling screen display</b>	185
Working with PAL canvases . . . . .	185
Canvas windows . . . . .	186
Paradox object canvases . . . . .	186
The full-screen canvas . . . . .	186
Showing the user the desktop . . . . .	186
Controlling local echo . . . . .	187
Specifying the current canvas . . . . .	188
Turning the canvas off and on . . . . .	191
Writing on a canvas . . . . .	191
Positioning the canvas cursor . . . . .	192
Displaying messages and text . . . . .	192
Formatting values . . . . .	193
Setting canvas colors and styles . . . . .	194
STYLE . . . . .	194
PAINTCANVAS . . . . .	195
Setting global and local colors . . . . .	195

Global colors . . . . .	195
The GETCOLORS command . . . . .	195
The SETCOLORS command . . . . .	200
The SYSCOLOR() function . . . . .	200
Local colors . . . . .	201
The WINDOW GETCOLORS command . . . . .	201
The WINDOW SETCOLORS command . . . . .	204
<b>Chapter 13</b>	
<b>Handling events</b>	205
What is an event? . . . . .	205
The categories of events . . . . .	206
Mouse events . . . . .	206
Key events . . . . .	206
Message events . . . . .	206
Idle events . . . . .	207
The event stream . . . . .	207
Mouse event routing . . . . .	207
Key event routing . . . . .	208
Message event routing . . . . .	208
Trapping and blocking events . . . . .	209
GETEVENT event trapping . . . . .	209
Localizing events . . . . .	210
WAIT event trapping . . . . .	211
SHOWDIALOG event trapping . . . . .	212
Event blocking . . . . .	213
The GETCHAR() function . . . . .	213
The ACCEPT command . . . . .	214
Typical application event processing . . . . .	214
SHOWPULLDOWN and GETMENSELECTION . . . . .	215
SHOWPULLDOWN and WAIT . . . . .	215
SHOWPULLDOWN and GETEVENT . . . . .	216
Valid events . . . . .	216
Valid mouse events . . . . .	216
Valid key events . . . . .	217
Valid message events . . . . .	217
Valid idle events . . . . .	218
Using triggers . . . . .	218
Trigger categories . . . . .	220
Trigger sequence . . . . .	220
WAIT triggers . . . . .	221
SHOWDIALOG triggers . . . . .	221

## Chapter 14

<b>Creating menus</b>	223
Displaying a menu	223
SHOWMENU command	224
Controlling the action	225
SHOWPOPOP command	226
SHOWPULLDOWN command	229
SHOWPULLDOWN and GETMENSELECTION	230
SHOWPULLDOWN and GETEVENT	232
SHOWPULLDOWN and WAIT	233
Disabling and enabling menu items	235
Removing a pull-down menu	236

## Chapter 15

<b>Creating dialog boxes</b>	237
Displaying a dialog box	238
Creating simple dialog boxes	238
SHOWFILES and SHOWTABLES dialog boxes	238
SHOWARRAY dialog boxes	240
Creating complex dialog boxes	241
Using control elements in a dialog box	242
Push buttons	243
Pick lists	244
PICKFILE control element	245
PICKTABLE control element	247
PICKARRAY control element	248
PICKDYNARRAY control element	250
PICKDYNARRAYINDEX control element	252
Radio buttons	253
Check boxes	255
Type-in boxes	258
Sliders	259
Labels	261
Using canvas elements to control dialog box background	262
Using the dialog procedure	263
Structure of the dialog procedure	264
Using commands and functions to control the dialog box	267
Using the REPAINTDIALOG command	268
Using the RESYNCCONTROL command	269
Using the RESYNCDIALOG command	272

Using the SELECTCONTROL command	274
Using the REFRESHCONTROL command	275
Using the REFRESHDIALOG command	277
Using the ACCEPTDIALOG command	279
Using the CANCELDIALOG command	280
Using the NEWDIALOGSPEC command	281
Using the CONTROLVALUE() function	283

## Chapter 16

<b>Using the WAIT command</b>	285
Interacting with the user	285
Two forms of the WAIT command	286
Keypress-driven WAIT	287
Event-driven WAIT	288
Using the WAIT procedure	289
Using a SHOWPULLDOWN menu	293
Setting Retval from the WAIT procedure	294
Table navigation during a WAIT	295
WAIT and multi-table forms	296
Using lookup help	298
Changing the event list during a WAIT	299
Changing the SHOWPULLDOWN menu with NEWWAITSPEC	300

## Chapter 17

<b>Using multi-table forms</b>	303
Basic concepts	303
Working with linked forms	304
Link keys	305
Restricted views	307
Multi-table forms and the Paradox workspace	310
Referential integrity	312
The blank key problem	313
Ensuring referential integrity	314

**Part IV**  
**Controlling Paradox** 315

**Chapter 18**  
**Interacting with Paradox** 317

- How modes determine what you can do . . . 317
  - The current mode . . . . . 319
  - Minor modes . . . . . 319
- Using recorded keypress interaction . . . 320
- Simulating keypress interaction . . . . 323
  - Abbreviated menu commands . . . . 323
  - SELECT, KEYPRESS, and TYPEIN . . . 323
- Working with tables . . . . . 325
  - Locating records . . . . . 325
  - Looking up and storing values . . . . 326
    - Field specifiers . . . . . 326
    - Arrays and record values . . . . . 327
- Checking the desktop status . . . . . 327
- Column calculations . . . . . 330

**Chapter 19**  
**Manipulating values** 333

- Field specifiers . . . . . 333
- Using variables . . . . . 334
  - Lifetime of variables . . . . . 335
  - Typing variables . . . . . 336
- Using fixed arrays . . . . . 337
  - Manipulating records in fixed arrays . . 337
  - Field names as subscripts . . . . . 339
- Using dynamic arrays . . . . . 340
- Using variables to compute commands . . 342
- Query variables . . . . . 343

**Chapter 20**  
**Keyboard macros** 347

- Defining macros with SETKEY . . . . . 347
  - Keycodes to use . . . . . 348
  - Commands to use . . . . . 348
- Using macros . . . . . 349
  - Using macros to reorder tables . . . . 350
  - Using macros to present information . . 350

- Using macros for convenience . . . . . 351
- Using macros wisely . . . . . 351
- Canceling macros . . . . . 352

**Chapter 21**  
**Performance and resource tuning** 353

- Scripts . . . . . 353
- Procedure and script swapping . . . . 354
- Managing memory in interactive Paradox 355
  - Swap devices . . . . . 356
  - Command-line options . . . . . 356
- Managing memory in an application . . 356
  - RMEMLEFT() and the code pool . . . . 357
  - Handling a code pool invasion . . . . 358
- Indexing tables . . . . . 359
  - How indexes are maintained . . . . . 360
  - Techniques . . . . . 361

**Part V**  
**Putting it all together** 363

**Chapter 22**  
**Building Paradox applications** 365

- How to create applications . . . . . 366
  - Design and create the building blocks . . 367
  - Design the application's structure . . . . 368
  - Record small components in scripts . . . 369
  - Write additional procedures . . . . . 370
  - Tie up the loose ends . . . . . 370
  - Use the Debugger to test your work . . . 370
  - Tune the application . . . . . 371
  - Create closed procedures . . . . . 371
- Packaging your application . . . . . 372
  - Script files . . . . . 372
  - Procedure libraries . . . . . 372
  - Paradox objects . . . . . 373
  - Paradox or Paradox Runtime . . . . . 373
  - Documentation . . . . . 374
  - Initial batch file . . . . . 374

**Chapter 23**

**Multuser applications** . . . . . 375

General principles . . . . . 375

  Converting single-user to multuser applications . . . . . 376

  Organizing a multuser application . . . . . 376

  Private directories . . . . . 377

Controlling access to data . . . . . 378

Managing locks . . . . . 378

  Automatic locking . . . . . 379

  Explicit locking . . . . . 379

  Table locks . . . . . 380

    Using LOCK and UNLOCK . . . . . 382

    Deadlock . . . . . 383

    Multiple locks on one table . . . . . 384

    Using LOCKSTATUS() . . . . . 384

  Record locking . . . . . 385

    How to lock records . . . . . 385

  Using LOCKRECORD and UNLOCKRECORD . . . . . 387

    Locking existing records . . . . . 387

  Using POSTRECORD . . . . . 387

    Handling key conflicts . . . . . 388

    Multiple record locks . . . . . 390

  Using multi-table forms on a network . . . . . 391

    Simultaneous locking of multiple tables . . . . . 391

    Group locks . . . . . 391

    Write record locks . . . . . 392

    Record locking in multi-table forms . . . . . 392

The SETRETRYPERIOD command . . . . . 393

Currency of data . . . . . 393

  Query and report integrity . . . . . 394

  Crosstab integrity . . . . . 395

Interprocess communication . . . . . 395

  Using the SETBATCH command . . . . . 396

**Chapter 24**

**The Sample application** . . . . . 397

What is the Sample application? . . . . . 397

How to use the Sample application . . . . . 398

Code listing and commentary . . . . . 401

  Code listing for SAMPLE.SC . . . . . 402

  Code listing for RPTALL.SC . . . . . 414

**Index**

429

2-1	Comments and blank lines . . . . .	19	16-3	Trigger sequences in a WAIT . . . . .	292
2-2	Types of PAL commands . . . . .	21	16-4	Basic table navigation during a WAIT	295
2-3	Recorded keypress interaction . . . . .	23	16-5	Table navigation with the WAIT procedure . . . . .	295
2-4	A query image in a script . . . . .	26	16-6	A simple multi-table WAIT procedure	297
3-1	Using tilde variables in queries . . . . .	41	16-7	Using default Paradox lookup help .	298
3-2	Referring to fields . . . . .	52	16-8	Changing the event list with NEWWAITSPEC . . . . .	299
3-3	Branching based on the next keystroke . . . . .	61	16-9	Changing a SHOWPULLDOWN menu during a WAIT . . . . .	300
4-1	IF...THEN...ELSE branching . . . . .	65	17-1	Using LINKTYPE to distinguish parts of a form . . . . .	306
4-2	SWITCH branching from a menu . . . . .	67	17-2	Using functions with restricted views	309
4-3	Dynamic array branching . . . . .	68	17-3	Functions on a multi-table form and the workspace . . . . .	311
4-4	Using WHILE to redisplay a menu . . . . .	72	18-1	Using status functions . . . . .	330
4-5	Scanning a table . . . . .	74	18-2	Calculating test statistics . . . . .	332
4-6	Interrupting a loop . . . . .	75	19-1	Manipulating array elements . . . . .	339
5-1	Using a picture to format input . . . . .	77	19-2	Manipulating dynamic array elements . . . . .	340
5-2	Options in pictures . . . . .	82	19-3	Simulating object-oriented programming . . . . .	340
5-3	Alternatives in a date picture . . . . .	83	19-4	Using variables to adjust a query .	345
6-1	Dynamic scoping . . . . .	103	20-1	Using macros to update records . . .	349
6-2	Using procedure libraries . . . . .	111	20-2	Presentation macros . . . . .	350
6-3	Creating a procedure library . . . . .	112	21-1	Indexing a table . . . . .	360
6-4	Reading and using library procedures . . . . .	113	23-1	Locking records . . . . .	389
7-1	Protecting scripts and tables . . . . .	117			
7-2	A sample <i>Init</i> script . . . . .	120			
7-3	Using error procedures . . . . .	125			
8-1	A miniscript macro . . . . .	133			
8-2	Screen display in a miniscript . . . . .	135			
9-1	A query form and its QuerySave image . . . . .	143			
9-2	A sample recorded script . . . . .	143			
10-1	A test script . . . . .	162			
10-2	Debugging a script . . . . .	163			
11-1	Floating window and desktop interactions . . . . .	183			
15-1	Dialog procedure arguments and trigger sequence . . . . .	266			
16-1	Basic table interaction in a WAIT . . .	287			
16-2	Using WAIT in a WHILE loop . . . . .	288			



# TABLES

<p>1-1 Printing conventions . . . . . 8</p> <p>1-2 Syntax notation . . . . . 8</p> <p>2-1 Special keys in scripts . . . . . 24</p> <p>2-2 Abbreviated menu commands . . . . . 28</p> <p>3-1 Resulting data type of mixed expressions . . . . . 33</p> <p>3-2 Backslash sequences . . . . . 35</p> <p>3-3 Valid and invalid naming conventions . . . . . 39</p> <p>3-4 PAL operators . . . . . 46</p> <p>3-5 Comparison rules . . . . . 49</p> <p>3-6 Order of precedence of PAL operations within expressions . . . . . 50</p> <p>3-7 Field specifiers . . . . . 51</p> <p>3-8 String functions . . . . . 55</p> <p>3-9 Date formats . . . . . 56</p> <p>3-10 Date functions . . . . . 57</p> <p>5-1 Match characters in pictures . . . . . 78</p> <p>5-2 Special features of pictures . . . . . 80</p> <p>5-3 Format specifications . . . . . 85</p> <p>7-1 Paradox error codes . . . . . 122</p> <p>10-1 Debugging keys . . . . . 155</p> <p>11-1 Window attributes . . . . . 175</p> <p>12-1 Attributes of screen elements 0–31 . . . 196</p> <p>12-2 Attributes of screen elements 1000–1077 . . . . . 198</p> <p>12-3 Local color attributes for windows . . . 202</p> <p>12-4 Local color attributes for dialog boxes . . . . . 202</p> <p>13-1 Mouse events . . . . . 206</p> <p>13-2 Message events . . . . . 207</p> <p>13-3 Additional elements returned by GETEVENT . . . . . 209</p> <p>13-4 Elements of a valid mouse event . . . 217</p> <p>13-5 Elements of a valid key event . . . . 217</p> <p>13-6 Elements of a valid message event . . . 217</p> <p>13-7 SHOWDIALOG triggers . . . . . 218</p> <p>13-8 SHOWDIALOG UPDATE triggers . . . . 219</p>	<p>13-9 WAIT triggers . . . . . 219</p> <p>15-1 Assignments to dialog procedure arguments made by events . . . . . 264</p> <p>15-2 Assignments to dialog procedure arguments made by triggers . . . . . 265</p> <p>15-3 RESYNCCONTROL actions vs. REFRESHCONTROL actions . . . . . 271</p> <p>16-1 WAIT procedure return values . . . . 291</p> <p>16-2 Values assigned to Retval by WaitProc procedure . . . . . 294</p> <p>17-1 Link keys in multi-table forms . . . . 306</p> <p>17-2 Restricted view commands and functions . . . . . 308</p> <p>17-3 Commands to navigate the workspace . . . . . 310</p> <p>18-1 Comparison of record location methods . . . . . 326</p> <p>18-2 PAL status commands and functions . . . 328</p> <p>18-3 Column calculation functions . . . . . 330</p>
---	---

10-1	Debugging levels . . . . .	154	15-14	Dialog box displaying current system time . . . . .	268
10-2	The Debugger screen . . . . .	155	15-15	Pick list and type-in box values are the same . . . . .	273
10-3	Finding out where you are . . . . .	160	15-16	Pick list displaying files in a user-specified directory . . . . .	276
11-1	Current image and current window . . . . .	178	15-17	Two pick lists in a user-specified directory . . . . .	278
11-2	Paradox z-order . . . . .	180	16-1	Trigger sequences in a WAIT . . . . .	292
11-3	Layers of the z-order . . . . .	181	19-1	Variables in queries . . . . .	343
12-1	Annotating an image window . . . . .	189	22-1	Typical application menu . . . . .	369
12-2	Current canvas, current image, and current window . . . . .	190	24-1	Sample application splash screen . . . . .	398
12-3	Canvas coordinates . . . . .	191	24-2	Sample application about box . . . . .	399
13-1	Event stream in a typical application . . . . .	215	24-3	Sample application <i>Order Entry</i> form . . . . .	399
14-1	SHOWMENU pop-up menu . . . . .	225			
14-2	SHOWPOPUP pop-up menu . . . . .	227			
14-3	SHOWPULLDOWN menu with an empty workspace . . . . .	230			
14-4	SHOWPULLDOWN menu with a window on the workspace . . . . .	232			
14-5	SHOWPULLDOWN menu with a WAIT table . . . . .	234			
15-1	Typical SHOWFILES or SHOWTABLES dialog box . . . . .	239			
15-2	SHOWFILES dialog box with pick list of scripts . . . . .	240			
15-3	SHOWDIALOG with push buttons . . . . .	243			
15-4	Two-column pick list of scripts . . . . .	246			
15-5	Two-column pick list of tables . . . . .	248			
15-6	SHOWDIALOG with PICKARRAY pick list . . . . .	249			
15-7	SHOWDIALOG with PICKDYNARRAY pick list . . . . .	251			
15-8	SHOWDIALOG with PICKDYNARRAYINDEX pick list . . . . .	253			
15-9	SHOWDIALOG with radio buttons . . . . .	254			
15-10	SHOWDIALOG with check boxes . . . . .	256			
15-11	SHOWDIALOG with slider and type-in box . . . . .	259			
15-12	SHOWDIALOG canvas elements . . . . .	263			
15-13	Typical values of dialog procedure arguments . . . . .	267			

# Introduction

This chapter introduces PAL, the Paradox Application Language. PAL lets you extend the power of Paradox to create database applications, no matter how complex, with a minimum of effort.

Using PAL, you can easily create applications that work and look exactly like Paradox. You can even develop self-contained applications that your users can work with effectively without knowing—or even owning—Paradox.

PAL is the most sophisticated and comprehensive of several tools that are available to customize Paradox to serve your application needs:

- ❑ If you want to automate tasks you perform regularly, you can record Paradox scripts that perform those tasks and play them back at any time.
- ❑ When you need to enhance the power or performance of recorded Paradox scripts, PAL is there to help.
- ❑ Paradox SQL Link lets you work with data stored in Structured Query Language (SQL) database servers. SQL Link provides new PAL commands and functions to help you manage the connection between Paradox and the database server. Once you install SQL Link, you have all the power of SQL at your fingertips, without ever needing to learn SQL programming. You can even embed SQL statements directly in your PAL applications. See the SQL Link *User's Guide* for details.
- ❑ The Paradox Engine is a library of C functions and Pascal procedures and functions that lets you write either entire Paradox applications or parts of applications in C or Pascal. This library of functions lets you manipulate Paradox tables in both single-user and multiuser environments.

This manual assumes at least a basic understanding of programming concepts and will not teach you how to program. If you've programmed in C, Pascal, or another programming or database

language, you probably understand enough to begin using PAL effectively.

---

## What is PAL?

PAL is a full-featured, high-level, structured database programming language that lets you write sophisticated Paradox programs (scripts) and applications. There are two aspects to PAL:

- the language itself: its commands, functions, and constructs
- a set of facilities for working with the language: a PAL menu, an Editor, a built-in debugger, and several ways of creating and playing scripts

---

## PAL and Paradox

As a developer or programmer of database applications, you can use PAL to take full advantage of all of the functionality of Paradox. In addition, you have access to a set of programming features and tools such as you'd expect in the most sophisticated application programming environments. Here are some of PAL's main features:

- Access the power of Paradox: You can think of PAL as a sort of robot Paradox user—an agent who operates Paradox for a user under your control. In PAL, the total power of Paradox is always available to you. Here are a few of the features of Paradox important to the application developer:
  - The Paradox query language with its powerful query-by-example (QBE) interface lets you construct queries, from the simplest to the most complex, with ease. The power to join many tables in a single query is already built into Paradox; PAL gives you the additional ability to incorporate variables into query statements. The unique capability of Paradox to save a query form makes it easy for you to use saved query statements in your PAL programs. (With Paradox SQL Link installed, you can use QBE to access data stored in SQL database servers.)
  - PAL has direct access to Paradox's extensive validity checking capabilities. Certain values, such as dates, are checked automatically. Other types of validity checks, including picture formatting, table lookups, and automatic fill-in of values between tables, can be established quickly at any point.

- You can create new table structures quickly as you need them; you can also modify existing table structures without risking data loss. Whenever you modify the structure of a table, Paradox automatically updates all objects associated with that table, such as forms, reports, and indexes.
- You can use Paradox's multi-table forms for sophisticated data entry and display applications, such as invoicing or order entry, that require one-to-many, many-to-many, and many-to-one relationships among records stored in different tables. You can use multi-record forms to automatically provide scrolling regions of detail records in these types of applications.
- PAL also gives you access to Paradox's sophisticated Report Designer, which lets you design a variety of output formats without programming. The Report Designer supports up to 16 levels of grouping. It also outputs calculated values based on expressions containing one or more fields. You can develop report specifications interactively onscreen in Paradox and then easily incorporate them into your PAL programs.
- You can enhance your PAL programs by accessing Paradox's built-in graphics, which make it easy to build decision support and analysis applications.
- Emulate the user interface of Paradox: You can design menu bars, pull-down menus, pop-up menus, and dialog boxes to present a friendly interface to your users. You can also create and manipulate Paradox windows. The user can interact with your application using the keyboard or the mouse.
- Define unlimited variables and arrays: In PAL, the number of variables and arrays you can define is limited only by system memory. Fixed arrays can contain up to approximately 15,000 elements. Built-in commands for using fixed arrays to manipulate an entire record at once let you move records between tables quickly.
- Use dynamic arrays: For maximum flexibility and power, PAL provides arrays that change size dynamically. You do not have to declare the number of elements in advance. Dynamic arrays can have an unlimited number of elements. Instead of a subscript, each element of the array has a unique tag. You access the value of the element through this tag as you would access the value of a fixed array element through its subscript. Special commands, functions, and control structures support access to and manipulation of these dynamic arrays.

- ❑ Powerful procedure concepts: PAL places no restrictions on the number of procedure definitions that can be active at one time. In addition, PAL procedures allow for:
  - ❑ private and global variables
  - ❑ recursion and nesting of scripts and procedures
  - ❑ dynamic scoping of variables
  - ❑ automatic memory management for procedures (loading them into and releasing them from memory)
  - ❑ procedure definition and usage comparable to other high-level languages such as C and Pascal
  - ❑ the ability to create and manage libraries of preparsed definitions
- ❑ A complete collection of built-in functions:
  - ❑ mathematical and statistical functions
  - ❑ financial functions, such as mortgage-payment calculation and computation of present and future values
  - ❑ string functions that allow full text manipulation, including searching and concatenation, for strings of characters up to approximately 64 megabytes
  - ❑ system functions that let you manipulate and determine the status of Paradox objects
- ❑ Event-driven programming: In addition to events caused by keyboard interactions, PAL gives you access to events caused by mouse interactions. PAL also puts you in charge of idle events generated by your system, triggers resulting from table interactions such as field departures, and message events resulting from interactions such as closing a window or accepting a dialog box.
- ❑ Extensive screen and keyboard I/O capabilities: PAL gives you complete control over all aspects of I/O and interaction with end users. With the Paradox custom forms, you can design data entry and output screens for your applications interactively (without programming). The WAIT command lets you give selective control to users so they can browse or update records, fields, or entire tables. When a user presses a designated key or mouse button, or causes a certain class of event, control returns to your script or to the next higher level of a WAIT.
- ❑ Control-of-flow commands: In addition to traditional control structures, such as IF...THEN...ELSE, WHILE...ENDWHILE, and SWITCH...CASE...ENDSWITCH, PAL has a

powerful SCAN construct that lets you perform sequences of operations on each record of a table very quickly.

- ❑ Easy-to-create secondary indexes: With PAL, you can set up secondary indexes on Paradox tables. You don't need to worry about them at development time; You can easily create them after the fact to improve performance. Once you have established these indexes, you don't have to maintain them—Paradox updates them automatically whenever necessary.
- ❑ Full password encryption: Through Paradox, you can selectively password protect data in both tables and scripts. You have control over user access down to the field level. You can also control access to forms, reports, and other objects.
- ❑ Computation macros: You can compute strings that are interpreted as commands.
- ❑ Abbreviated menu commands: PAL gives you a special class of commands that lets you emulate the Paradox menu selection process using syntax and parameters, in a style of a conventional programming language.
- ❑ Multiuser capabilities: You can take full advantage of Paradox's built-in file and record locking capabilities, as well as an extensive set of commands for explicit locking.
- ❑ Referential integrity: When you use multi-table forms, Paradox automatically maintains referential integrity between linked tables.
- ❑ Error handling: You can define special procedures to trap and process run-time script errors.

---

## **PAL facilities**

The PAL language is built into Paradox, along with an integrated set of facilities for working with it, including

- ❑ PAL menu
- ❑ PAL Editor
- ❑ PAL Debugger
- ❑ script recording and playing

These facilities form a powerful, integrated environment for prototyping, developing, testing, tuning, and playing scripts and applications.

You might also want to order Paradox Runtime, which lets you provide your application to users who don't own Paradox.

Although the PAL environment provides unparalleled support for application development, you don't have to use these facilities to

create PAL programs. Since PAL programs start as ASCII text files, you can use your own editor to create them if you want (in fact, you can easily integrate your favorite editor into the PAL environment).

---

## Why use PAL?

The PAL environment streamlines the process of creating, modifying, and maintaining applications. PAL can quickly put together complete applications with minimal programming effort.

Programming in this environment is a three-step process:

1. First use Paradox interactively to create the tables, queries, forms, reports, and other objects needed for an application.
2. Then record scripts to capture interactive operations. Use PAL to build a menu structure and define relationships among the elements of your application.
3. Use the Editor, Debugger, and other PAL facilities to enhance and fine-tune the application.

With Paradox and PAL, you spend much of your time using Paradox interactively to design the components of your applications and comparatively less time writing PAL programs directly.

---

## Before you use PAL

PAL and Paradox are highly integrated. Therefore, the more you know about Paradox, the more you can take advantage of it in your PAL applications.

Throughout this manual, we assume you've read the *User's Guide* and you have experience using Paradox interactively. You should understand how to

- choose menu commands and how the various Paradox menus and modes interrelate
- choose options and settings in a dialog box
- create tables, forms, and reports
- construct query statements using QBE
- sort tables
- use the keyboard and mouse with Paradox



If you don't have this basic familiarity, spend more time with Paradox before trying to program with PAL.

If you've programmed applications with other databases, it would be natural to assume that you could jump in and start writing PAL scripts without understanding Paradox. After all, PAL is just another database programming language, right? Wrong.

As you gain familiarity with Paradox, you will begin to appreciate its power and feel more comfortable putting it to work for you. You will find that you can develop applications in a less constrained, more free-form way. *The better you understand Paradox, the better you'll understand PAL.*

---

## How to use this manual

This book is a comprehensive reference to the elements, commands, and features of the PAL language. Although it is not a tutorial, you will find plenty of script examples in it, along with advice on practical techniques for getting the most out of PAL. This manual is divided into five parts:

- Part I, "Basic building blocks," describes the basic elements of PAL, including scripts and commands, expressions, variables and arrays, control structures, and procedures.
- Part II, "PAL facilities," describes the tools you use to work with PAL, including the PAL menu, facilities for creating and playing scripts, the Editor, and the PAL Debugger.
- Part III, "Interacting with the user," describes using windows, controlling the screen display, handling events, and creating the various elements of the user interface, such as pull-down menus, pop-up menus, and dialog boxes. This part also describes how to use the WAIT command to pause processing while a user interacts with a table, a form, or a multi-table form.
- Part IV, "Controlling Paradox," describes the modes of Paradox, the workspace, value manipulation, and keyboard macros.
- Part V, "Putting it all together," describes how to build applications and uses an executable PAL application (included on your Paradox distribution disks) as a teaching tool. These chapters describe the design of a complete application, point out some of the techniques used to create one, and present the fully documented, structured code of the Sample application. The Sample application is installed in the PDOX40\SAMPAPP directory.

This book is intended to accompany Paradox version 4.0. You can use it to develop scripts and applications for previous versions of Paradox, but some of the programming techniques described here are only relevant for Paradox 4.0. Commands and functions that were implemented in this version of Paradox and in earlier versions are listed in Appendix I of the *PAL Reference*. When using these commands in an application, remember that they will cause errors if the application is run on a version of Paradox that doesn't support them.

## Printing conventions

Table 1-1 lists the typographical conventions used in this book.

Table 1-1 Printing conventions

Convention	Applies to	Examples
<b>Bold</b>	Any message displayed by Paradox	Paradox displays <b>Creating table...</b>
<i>Italic</i>	Tables, scripts, procedures, arrays, variables, glossary terms, emphasis, example elements	<i>Orders table, DoWait procedure, the Retval variable, array a</i>
ALL CAPS	DOS files and directories, reserved words, operators, commands	PARADOX.EXE, LIKE, SORT
Initial Caps	Applications, libraries, fields	Sample application, Price field
<i>Keycap font</i>	Keypress	<i>Enter, Home</i>
Monospaced font	Code examples	SETKEY "F1" SORT TABLE() ON FIELD()
<b>Type-in font</b>	Text that you type in	<b>=6/2/90, S@@@n</b>

PAL commands and functions use syntax notation. This notation is described fully at the beginning of the *PAL Reference* and summarized here:

Table 1-2 Syntax notation

Convention	Element	Examples	Meaning
ALL CAPS	Keyword	SHOWMENU	Type exactly as shown
<i>Italic</i>	Fill-in	<i>TableName</i>	Replace with expression
	Choice	{ VARS   PROCS }	Choose <i>one</i> of the elements separated by the vertical bar
[ ]	Optional	[ OTHERWISE ]	You <i>can</i> choose whether or not to include this

Convention	Element	Examples	Meaning
{ }	Required	{ VARS   PROCS }	You <i>must</i> choose one of the elements separated by the vertical bar

The fill-in possibilities are listed in “PAL variable parameters” in the *PAL Reference*. When you see an italicized word in a syntax definition, look it up in that table to find out what kind of expressions are allowed.

---

## Standard and compatible modes

If you are familiar with previous versions of Paradox, the first thing you’ll notice about Paradox 4.0 is its new interface. Most of the elements of the interface, including dialog boxes, pull-down menus, pop-up menus, and windows, are new in Paradox 4.0.

Although the interface has changed significantly, Borland has taken care to ensure that applications you developed in previous versions of Paradox will continue to run as well as possible in Paradox 4.0. Because Paradox 4.0 is a *windowing* environment, however, existing applications may not look appropriate under the new interface.

To accommodate existing applications, Paradox 4.0 supports two interfaces:

- *Standard* mode refers to the new windowing interface of Paradox 4.0.
- *Compatible* mode simulates the interface of Paradox 3.5 and lets you run existing applications so they will have their existing look and feel.

---

### Preparing existing applications for compatible mode

This section gives experienced PAL programmers the necessary information to prepare existing applications for Paradox 4.0 compatible mode.

To prepare existing applications to run in both modes, you must convert your libraries to Paradox 4.0 file format. After you convert your PAL libraries, existing applications should run unmodified in compatible mode. Once your users become familiar with the capabilities of Paradox 4.0, however, you will want to begin to modify your existing applications to take advantage of the new interface, the new features such as memo fields and secondary indexes, and the power of event-driven programming. For additional information about working with existing applications, contact Borland Customer Support.

The following instructions describe how to prepare existing applications for Paradox 4.0 compatible mode. The PAL commands used in this procedure are discussed in detail in the *PAL Reference*. Review the CONVERTLIB and SETUIMODE commands in the *PAL Reference* before you begin. See Chapter 23 of the *User's Guide* for information about converting existing tables to Paradox 4.0 file format.

To prepare your applications,

- Important**
1. Back up all your application files, including tables and their families, scripts, and libraries. (Always back up your files before performing any procedure that can affect your data.)
  2. Copy all the files for your existing application into a working directory.
  3. Start Paradox and change directories to your working directory.
  4. Convert all existing library files to Paradox 4.0 format.

Press PAL Menu *Alt-F10* to display the MiniScript type-in box and enter the command

```
CONVERTLIB "OLDLIB" "NEWLIB"
```

where *OLDLIB* is the name of a Paradox 3.0 or 3.5 library file and *NEWLIB* is the name of your new library file. Specify *OLDLIB* and *NEWLIB* without the .LIB extension. *OLDLIB* and *NEWLIB* cannot be identical. If *NEWLIB* already exists, it will be overwritten without warning.

- Optional**
5. Convert any existing tables to Paradox 4.0 file format. You do not have to convert your tables; however, converting tables will result in improved performance. After you convert tables, you will no longer be able to access those tables in Paradox 3.5.

Select Modify | Restructure from the Main menu, choose your table, and press *Return* to enter Restructure mode.

From the Restructure mode menu, select FileFormat | Standard to specify Paradox 4.0 file format.

Select DO-IT! from the Restructure mode menu to modify the file format of your table.

6. After you change the format of your libraries (and optionally, tables), exit Paradox and return to DOS.
7. Delete your Paradox 3.5 library files by issuing the DOS command

```
DEL OLDLIB.LIB
```

where OLDLIB.LIB is your 3.5 library file.

8. Rename your newly converted Paradox library file to the name that your 3.5 library file used by issuing the DOS command

```
REN NEWLIB.LIB OLDLIB.LIB
```

9. To run your application, start Paradox, press PAL Menu *Alt-F10* to display the MiniScript type-in box, and enter the command

```
SETUIMODE COMPATIBLE
```

10. After your application finishes running, enter the command

```
SETUIMODE STANDARD
```

to place Paradox in standard mode again.

This procedure is all that is necessary to prepare most applications to run in Paradox 4.0 compatible mode. You may eventually want to modify your scripts to include the SETUIMODE commands that specify the Paradox interface.

---

## **New commands and functions in Paradox 4.0**

Approximately 100 commands and functions have been added to Paradox 4.0, significantly enhancing its power and usability.

Many of the commands and functions added to Paradox 4.0 allow a PAL programmer to create the new interface elements, such as dialog boxes, menus, and windows. The commands and functions that create and support these interface elements are not available in compatible mode.

Other commands and functions were added to give a PAL programmer access to event-driven programming. Because versions of Paradox prior to 4.0 are keypress-driven, these commands and functions are not available in compatible mode.

Many of the new commands and functions can be used in compatible mode. The complete list of commands and functions added to Paradox 4.0 is in Appendix I of the *PAL Reference*. Commands and functions that are not available in compatible mode are indicated with an asterisk (\*).

---

## **Commands and functions prior to Paradox 4.0**

Where appropriate, the behavior of commands and functions existing in versions of Paradox prior to 4.0 is discussed in the *PAL Reference* and the *PAL Programmer's Guide*.

In some cases, commands and functions existing in versions of Paradox prior to 4.0 behave differently in compatible and standard modes. For example, the SHOWMENU command creates the expected ring-style menu in compatible mode; however, SHOWMENU creates a pop-up menu in standard mode.

You may find that existing applications will run in standard mode; however, those applications may make assumptions about the interface that are not true in standard mode. For example, standard mode lets users resize windows and move freely between them in ways that were not possible in previous versions of Paradox.

Until you understand the programming environment of Paradox 4.0 standard mode, consider running existing applications in compatible mode.

# Basic building blocks

This part of the *PAL Programmer's Guide* contains a description of the elements and constructs of the Paradox Application Language (PAL). Although PAL has its share of unique properties and concepts, you'll find many of its features familiar and common to other programming languages.

Part I consists of six chapters:

- Chapter 2, "Scripts and commands." A PAL program, or *script*, is made up of a series of commands. This chapter describes the general structure and syntax for PAL commands.
- Chapter 3, "Expressions." Expressions are arguments to PAL commands. This chapter describes the elements of expressions, including data values (constants), variables and arrays, operators, field specifiers, and functions. It also explains how to manipulate strings, dates, and keycodes in expressions.
- Chapter 4, "Control structures." This chapter describes the PAL commands used to alter the flow of execution in a script.
- Chapter 5, "Formatting data." This chapter explains how to use pictures and format specifications to control the way values are input from the keyboard and output to the screen or printer.
- Chapter 6, "Procedures." This chapter explains how to use procedures and procedure libraries to create your own custom functions and enhance the performance of your application.
- Chapter 7, "Special topics." This chapter covers naming conventions, password protection, the special scripts *Init* and *Instant*, and error processing.

The information in these chapters, along with the individual descriptions of PAL commands and functions in the *PAL Reference*, comprises a complete description of the features of PAL. Parts II, III, IV, and V help you use PAL commands and functions in building complete applications.





# Scripts and commands

A PAL program, or *script*, is made up of a series of statements. This chapter describes the structure of scripts and statements, and covers

- the nature of a script
- statement sequence
- command and function syntax
- types of PAL commands
  - programming commands
  - keypress interaction
  - abbreviated menu commands

Only the general syntax and structure of PAL commands are covered here. The specific syntax and use of each command are described in the command reference in the *PAL Reference*.

---

## The nature of a script

A PAL *script* consists of a sequence of commands. You can put together a script or series of scripts to perform one or more tasks or a complete database *application*—a single unit with which users can enter, view, maintain, and report on their data.

*Commands* specify actions for Paradox to perform. As you develop your scripts and applications, remember that Paradox, not PAL, is the actor who performs a script. You can think of PAL as an automated Paradox user that you control. This automated user guides the real user of your application through menus and operations, while hiding the parts of Paradox your users don't need to see.

PAL scripts are simply ASCII files with the extension .SC. Since scripts consist of ASCII text, you can use any ASCII text editor to

create them. Although you can link your favorite text editor into the Paradox environment, for efficiency and convenience it's often beneficial to use Paradox's Editor and built-in script recording methods to create scripts. See Chapter 9 for details on these and other ways of creating and playing scripts.

Unless you use branching or control structures to alter the flow of control, commands in a PAL script are executed in sequence from beginning to end. The following recorded script, for example, executes sequentially from the first command to the last:

```
MENU {View} {Orders} EDITKEY END DOWN MENU {Image} {PickForm} {1}
```

When the last command in a script has been executed, the script ends.

Scripts can also contain instructions to play other scripts. For instance, this is a valid script by itself, even though all it does is play another script:

```
PLAY "Script2"
```

A script that calls another script is temporarily suspended while the other script is played. The called script, in turn, can be suspended while other scripts are called. When each script finishes playing, control returns to the script that called it, which resumes where it left off. When the original script is finished, control returns to Paradox (you can change this as described in Chapter 4).

Experienced programmers may wonder how PAL scripts differ from traditional programs. Actually, they are quite similar in many ways:

- ❑ Scripts can be structured. Besides their normal beginning-to-end sequence of execution, PAL supports control structures and loops, such as WHILE...ENDWHILE, IF...THEN...ELSE...ENDIF, and SWITCH...CASE...ENDSWITCH, that let you prescribe an ordered structure of command execution.
- ❑ Much as in Pascal and C, you can define *procedures* to perform one or more tasks. Once you define a procedure (or load it from a library), you can use it in commands. Procedures can receive arguments from, and return results to, the script or procedure that calls them.
- ❑ Scripts and procedures can be nested within other scripts and procedures. PAL's procedure capabilities let you build subroutines within your applications. You can modularize an application by breaking it into single-function scripts controlled by a master script. You can nest over 65 thousand procedures and up to approximately 30 scripts.
- ❑ Also, as in C, you can freely use whitespace (tabs, spaces, and blank lines). You can choose to indent subordinate script lines or not; you can put one or more commands on a line, or you can

append a comment to any script line. Whitespace has no effect on how commands are executed.

For example, these two scripts execute exactly alike, but one of them is much easier to read than the other.

```
IF ISTABLE("Choices")           ; confusing formatting
= True THEN DELETE
"Choices" ELSE MESSAGE
"Choices doesn't exist"
SLEEP 2000 ENDIF
```

```
IF ISTABLE("Choices") = True    ; preferred formatting
THEN DELETE "Choices"
ELSE MESSAGE "Choices doesn't exist"
SLEEP 2000
ENDIF
```

PAL scripts let you control exactly what the user sees. As you'll see in Chapter 12, scripts normally hide what is happening on the Paradox workspace while the user sees a screen called the PAL *canvas*. PAL includes a full set of capabilities to manipulate what the user sees:

- commands to manipulate what's on the PAL canvas
- commands to show the Paradox workspace
- commands used to mimic Paradox operations, such as the Paradox menu system

As a PAL programmer, you have a wealth of tools and functions available, many of which should be familiar to you if you've programmed before.

---

## Commands

---

### Syntax

PAL scripts have a free-form structure. With few exceptions, you can put as many commands on a single line as you want, each separated by at least one space. You can also split a command between two or more lines, as long as you do not make the split in the middle of a keyword, name, or data value (such as a number or character string). Each line in a PAL script can be up to 132 characters long.

*Legal line splits*

```
IF
myVariable
= 24
THEN
DoSomeProc()
ENDIF
```

*Illegal line splits*

```
IF my
Variable           ; whoops, split in middle of variable name
= 2
4                 ; no, value split in middle
THEN DoSomeProc
```

```
( )           ; no, procedure name and argument list
              ; must stay together

END
IF           ; no, keyword split in half
```

For certain commands (including RETURN, STYLE, SETKEY, ?, ??, and TEXT), there are restrictions on what can follow the command on the line. These restrictions are explained in the description of each command in the *PAL Reference*.

Many of the Paradox script recording methods (described in Chapter 9) create scripts with multiple commands on a line. For readability and ease of editing, however, you should place each command on a separate line and use indentation to show which commands are dependent upon others (especially control structures like IF...THEN).

---

### **Case**

Just as the structure of a script is free-form, so is the use of case in commands. Command keywords can be spelled in uppercase or lowercase letters or in any mixture of the two. For example, PAL recognizes Add, ADD, add, and even aDd as the same command. The same holds true for names of tables, fields, variables, arrays, and procedures. In fact, the only place case really matters in your scripts is in string (alphanumeric) data values. For example, the string "abc" is not equal to "ABC".

---

### **Comments and blank lines**

Unless it is contained in a string value, anything following a semicolon (;) on a script line is considered a comment and is ignored by PAL (see the following example). We recommend using comments to describe what is happening in the program; to identify cryptic messages, variables, or procedures; to provide any other information that might be useful to someone reading or editing the script; or simply to remember what it was you did. If you begin a line with a semicolon, the whole line will be a comment. Multi-line comments must have a semicolon at the beginning of each line.

Blank lines have no effect in PAL scripts. You can use them to set off comments and make your scripts more readable. A blank line does not require a semicolon.

## Example 2-1 Comments and blank lines

```
                                ; This script deletes a table
WINDOW CREATE @3,5 WIDTH 35 HEIGHT 5 TO InputWindow
@1,1 ?? "Table to delete: "      ; prompt for table name
ECHO NORMAL                      ; show window
ACCEPT "A8" to DelTable
WINDOW CLOSE
ECHO OFF
```

Blank line separates IF structure for readability

```
IF ISTABLE(DelTable) = True      ; test to make sure table exists
  THEN DELETE DelTable           ; if so, delete table
  ELSE MESSAGE "Table doesn't exist"
    SLEEP 2000
    MESSAGE ""
ENDIF
```

Semicolon separates comment from rest of line; Comment describes what the program is doing

## Sequence of execution

No special commands are needed to begin or end a script. When a script is played, its commands are simply executed in sequence from the beginning to the end. However, you can use certain control structures (like IF and WHILE) to alter the flow of control.

The play of a script ends either when all its commands have been executed, or when a command that terminates script execution—like RETURN, QUIT, or EXIT—is encountered. Where control passes when a script ends depends on how it is originally called and how it is ended:

- If the script is called or played by another script control returns to the next higher-level script (the script that played the one that ended).
- If the script is not played by another script or if it ends with a QUIT command, control returns to Paradox with the screen showing the current Paradox state.
- If the script ends with an EXIT command or is invoked with the Paradox Runtime program, control returns to DOS.

For example, when you play the script

```
VIEW "Orders" EDITKEY
```

it places the *Orders* table on the workspace, presses Edit *F9*, and then ends, leaving Paradox in Edit mode. As you'll see in Chapter 9, scripts can be played directly from the DOS command line, but they still return to Paradox when they finish. If you want to return to DOS from a script, use the EXIT command instead of QUIT.

The effects of PAL commands on objects on the Paradox workspace are normally hidden from the user until the script is finished playing or until you explicitly display the workspace with the ECHO command. Since operations on workspace objects are generally hidden while a script is playing, users normally see only the finished

results of the script, not any intermediate steps. You have complete control of what the user sees, however, and you can use commands to

- display menus (SHOWPULLDOWN, SHOWMENU, SHOWARRAY, and SHOWPOPUP)
- display prompts and messages (PROMPT, MESSAGE, RETURN, ?, and ??)
- accept user input (ACCEPT, SHOWDIALOG, WAIT)
- show all action while a script is being played (ECHO).

---

### **Scripts within scripts**

If you are programming complex tasks, you will probably find it more convenient and efficient to create a series of small scripts rather than one large one. You can use the PLAY command in one script to play a second script, from that script you can PLAY a third, and so forth.

Another way to use subroutines within an application is to create a procedure for each task or subroutine. Because procedures are modular and limited to specific tasks, they are easier to debug than scripts that are not composed of procedures. After you debug your individual procedures you can store them in a library. See Chapter 6 for complete information about procedures and libraries.

Scripts played within other scripts should *not* use the QUIT or EXIT command, unless you specifically want to terminate all script play at that point.

---

## **Types of PAL commands**

There are three major categories of PAL commands:

- *Programming commands* and functions accomplish things you can't do directly with Paradox. The most important of these extensions are control structures such as IF and WHILE (described in Chapter 4).
- *Keypress interactions* can be recorded from within Paradox. These commands and keystrokes represent Paradox actions one-for-one and include
  - direction keys
  - braced menu selections, such as {View}
  - special key commands, such as MENU and PGUP
  - quoted strings, such as "Philo T. Farnsworth"

## Programming commands and functions

- query forms saved with Scripts | QuerySave
- *Abbreviated menu commands* emulate Paradox menu commands, such as VIEW and PLAY. Not all Paradox menu commands can be abbreviated in this way, but the ones used most often can be.

You can see examples of each of these categories of commands in Example 2-2.

PAL programming *commands* and *functions* let your scripts do things they couldn't do with Paradox alone, such as display custom menus and repeat actions an arbitrary number of times. These commands can't be recorded from Paradox; you must use the Editor or your own editor to add them to a script.

PAL programming commands consist of one or more *keywords* (the command itself) and possibly some *parameters* (variable values, also known as *arguments*). For example, the BEEP command, which sounds a beep, consists of the single keyword BEEP. The MESSAGE command, on the other hand, takes a comma-separated list of string arguments, as in:

Command MESSAGE "The time is: ", TIME()  
 First argument Second argument

Arguments passed to PAL commands or functions are called *expressions* and are described in detail in Chapter 3.

In addition to performing an action, PAL functions differ from PAL commands because they return a value. For instance, if you enter the statement `x = SomeFunction()`, then the value that `SomeFunction()` returns will be assigned to `x`. Functions often take arguments, but not always. If a function does take an argument, that argument is entered between the parentheses that follow the function. A set of parentheses must always follow a function regardless of whether that function actually takes an argument.

### Example 2-2 Types of PAL commands

Here's a PAL script that lets the user choose a Quattro Pro spreadsheet, imports it into a Paradox table, sorts it by the first field, and titles and prints a standard report on it.

Programming commands {

```

SHOWFILES NOEXT                                ; display menu of WQ1 files
"*.*wq1"
"Select a QUATTRO PRO spreadsheet to report on."
TO SSName

IF SSName = "Esc"                               ; if user escapes from menu
  THEN QUIT                                     ; then quit to Paradox, otherwise
  ELSE ImportTbl = SSName                       ; assume table has same name
ENDIF
  
```

```

IF ISTABLE(ImportTbl)                ; if table exists
THEN WINDOW CREATE @20,5 WIDTH 45 HEIGHT 5 TO InputTable
  @1,1 ?? ImportTbl, " table exists. Enter new table name: "
  ECHO NORMAL
  WHILE ISTABLE(ImportTbl)
    ACCEPT "A8" TO ImportTbl ; prompt for new table name
  ENDWHILE \
  WINDOW CLOSE
  ECHO OFF
ENDIF

Braced menu commands ----- MESSAGE "Importing spreadsheet..."
                             {Tools} {ExportImport} {Import} ; import spreadsheet
                             {Quattro} {2} QUATTRO PRO} ; from Quattro Pro into
                             SELECT SSName SELECT ImportTbl ; table of name specified

Quoted string ----- MESSAGE "Sorting table..."
                      RIGHT ; move to first field
                      SORT ImportTbl ON FIELD() ; sort table on current field

Special key commands ----- MESSAGE "Designing and printing report..."
                             MENU {Report} {Design} ; design a report
                             SELECT ImportTbl {1}
                             SELECT "Report on " + ImportTbl ; change report title
                             {Tabular}
                             DO_IT! ; end report design mode

Abbreviated menu commands ----- REPORT ImportTbl "1" ; print the new report

```

## Keypress interactions

Four kinds of PAL commands let you reproduce the effect of any keypresses you can make when using Paradox interactively:

- ❑ menu commands, including choices from Paradox menus and typed responses to prompts in the menu
- ❑ Paradox special keys, including keys that have special meanings in Paradox, such as Do-It! *F2*, Down ↓, and Rotate *Ctrl-R*
- ❑ quoted strings, including anything you type outside of a menu (such as entering data in a table)
- ❑ query images, which are text representations of query statements

You'll find examples of these elements in Example 2-2 through Example 2-4. Although you can always type any of these elements directly into a script, it's usually easier to record them during an interactive Paradox session using Paradox's built-in script recording capabilities. To do so, use Scripts | BeginRecord from the Main menu or the PAL menu, or Instant Script Record *Alt-F3* (see Chapter 8).

## Menu choices

When a Paradox menu is displayed, your script can make a choice from it by enclosing the name of the menu command in braces { }. Similarly, when a prompt is displayed in the menu area, your script can enter the response by placing it in braces. {Modify} and {Orders} in Example 2-3 are menu choices.



In a recorded script, menu commands appear in initial capital letters, exactly as they appear on the menu. Responses to prompts are recorded exactly as typed. The case of the commands doesn't matter, however. You can use capital or lowercase letters when you use an editor to insert braced menu commands in a script. Although you can't use a variable name within curly braces, you can use `SELECT` to achieve the same effect—typing an arbitrary choice when the script is executed. See “Simulating keypress interaction” in Chapter 18 for details.

---

## Special keys

You can reproduce the effect of Paradox special keys or key combinations in a script by referencing their names (`Down` and `Do_It!` in Example 2-3).

Table 2-1 shows the name of each special key as it would appear in a script. The case (upper or lower) of the names is not important. Notice that `Do-It! F2`, whose name contains a hyphen, is recorded as `Do_It!` (with an underscore) in scripts. (If you type `Do-It!` with a hyphen in a script, PAL thinks you want to subtract the variable `It!` from the variable `Do`.) Keys with long, multi-word names, like `Instant Script Play Alt-F4` are compressed into a single word, as in `InstantPlay`.

Although you can't use a variable name to represent a key directly, you can use the `KEYPRESS` command to achieve the same effect—that of pressing an arbitrary key when the script is executed. You can also create a dynamic array to hold a key, and then use `EXECEVENT` to execute the keypress. See “Simulating keypress interaction” in Chapter 18 for details.

### Example 2-3 Recorded keypress interaction

---

Here's a recorded script that sorts the sample `Orders` table by date, with recent orders first.

If you know Paradox, you can easily understand how this script works. The first line contains braced menu choices that display a sort form for the `Orders` table. In the second line, the `↓` key is pressed four times to move the cursor to the `Date` field and a quoted string types the sort specification. Then `Do_It! F2` carries out the sort.

```
{Modify} {Sort} {Orders} {Same} ----- Braced menu choices
Down Down Down Down "1d" Do_It!
```

Special keys                      Quoted string

Table 2-1 Special keys in scripts

<b>Keys pressed</b>	<b>Key name in scripts</b>
<b>Cursor keypad and special keys</b>	
<i>Home</i>	Home
<i>End</i>	End
<i>PageUp</i>	PgUp
<i>PageDown</i>	PgDn
<i>LeftArrow</i>	Left
<i>RightArrow</i>	Right
<i>UpArrow</i>	Up
<i>DownArrow</i>	Down
<i>Insert</i>	Ins
<i>Delete</i>	Del
<i>Backspace</i>	Backspace
<i>Escape</i>	Esc
<i>Enter</i>	Enter
<i>Tab</i>	Tab
<i>Shift-Tab</i>	ReverseTab
<b>Cursor keys with Ctrl</b>	
<i>Ctrl-Break</i>	CtrlBreak
<i>Ctrl-Home</i>	CtrlHome
<i>Ctrl-End</i>	CtrlEnd
<i>Ctrl-LeftArrow</i>	CtrlLeft
<i>Ctrl-RightArrow</i>	CtrlRight
<i>Ctrl-Backspace</i>	CtrlBackspace
<i>Ctrl-PageUp</i>	CtrlPgUp
<i>Ctrl-PageDown</i>	CtrlPgDn
<b>Special Ctrl and Alt sequences</b>	
<i>CrossTab Alt-X</i>	CrossTab
<i>Ditto Ctrl-D</i>	Ditto
<i>DOS Ctrl-O</i>	DOS
<i>DOSBig Alt-O</i>	DOSBig
<i>Field View Ctrl-F</i>	FieldView
<i>KeyLookup Alt-K</i>	KeyLookup
<i>Lock Toggle Alt-L</i>	LockKey
<i>Rotate Ctrl-R</i>	Rotate
<i>Refresh Alt-R</i>	Refresh
<i>Delete Word Alt-D</i>	DeleteWord

<b>Keys pressed</b>	<b>Key name in scripts</b>
Delete Line <i>Ctrl-Y</i>	DeleteLine
Resync Key <i>Ctrl-L</i>	ReSyncKey
Undo <i>Ctrl-U</i>	Undo
Vertical Ruler Toggle <i>Ctrl-V</i>	VertRuler
Zoom <i>Ctrl-Z</i>	Zoom
Zoom Next <i>Alt-Z</i>	ZoomNext
<b>Function keys</b>	
Help <i>F1</i>	Help
Do-It! <i>F2</i>	Do_It! *
Up Image <i>F3</i>	UpImage
Down Image <i>F4</i>	DownImage
Example <i>F5</i>	Example
Checkmark <i>F6</i>	Check
Form Toggle <i>F7</i>	FormKey
Clear Image <i>F8</i>	ClearImage
Edit <i>F9</i>	EditKey
Menu <i>F10</i>	Menu
* Note difference between hyphen and underscore in Do_It!	
<b>Function keys with Alt</b>	
Instant Script Record <i>Alt-F3</i>	InstantRecord
Instant Script Play <i>Alt-F4</i>	InstantPlay
Field View <i>Alt-F5</i>	FieldView
Check Plus <i>Alt-F6</i>	CheckPlus
Instant Report <i>Alt-F7</i>	InstantReport
Clear All <i>Alt-F8</i>	ClearAll
CoEdit <i>Alt-F9</i>	CoEditKey
<b>Function keys with Ctrl</b>	
Window Next <i>Ctrl-F4</i>	WinNext
Window Resize <i>Ctrl-F5</i>	WinResize
Check Descending <i>Ctrl-F6</i>	CheckDescending
Graph Key <i>Ctrl-F7</i>	GraphKey
Window Close <i>Ctrl-F8</i>	WinClose
To QPro <i>Ctrl-F10</i>	ToQPro
<b>Function keys with Shift</b>	
Window Maximize <i>Shift-F5</i>	WinMax
GroupBy <i>Shift-F6</i>	GroupBy

---

## Quoted strings

In a script, enclose typed text in double quotation marks (""). You need to do this to enter or edit data in a table, type literals on a form or a report specification, fill out sort forms, and in many other situations. An example is the "1d" entered in the sort form in Example 2-3.

Note the difference between a typed response to a menu prompt, such as {Orders}, and typing outside the menu area, such as "1d".

Although you can't use a variable name within a quoted string, you can use the TYPEIN command to achieve the same effect—varying what is typed when the script is executed. See "Simulating keypress interaction" in Chapter 18 for details.

---

## Query images

To execute a query in a script, use Paradox to construct the query, and then choose Scripts | QuerySave from the Main menu to record it. This records a text representation of the query statement and embeds it between the QUERY command and the ENDQUERY keyword. Example 2-4 shows the correspondence between a query statement as seen onscreen and its representation in a QuerySave script.

Once you've saved the query, use the File | InsertFile command in the Editor to embed the query script in your script. Or, simply use the PLAY command to play the query script from your own script.

**Note** QUERY only *displays* the query statement on the workspace. To execute it, follow ENDQUERY with DO\_IT!

You can use *tilde variables* in query statements to dynamically pass values to the query before it is executed. See "Query variables" in Chapter 19 for details.

While it is possible to create a QUERY...ENDQUERY directly with an editor, we strongly suggest that you don't. When you construct a query statement interactively, Paradox provides structured support for entering and changing values in the query forms you're using. It also checks your queries for acceptability. Creating them with an editor bypasses Paradox's support and checks and can result in queries that look correct but do not execute.

For more information about recording a query script, see the discussion of Scripts | QuerySave in Chapter 9.

---

### Example 2-4 A query image in a script

Suppose you query the *Orders* and *Customer* tables for customers who ordered more than one of some item. Here is the image that is recorded in the *QuerySave* script:

Image recorded in  
QuerySave script

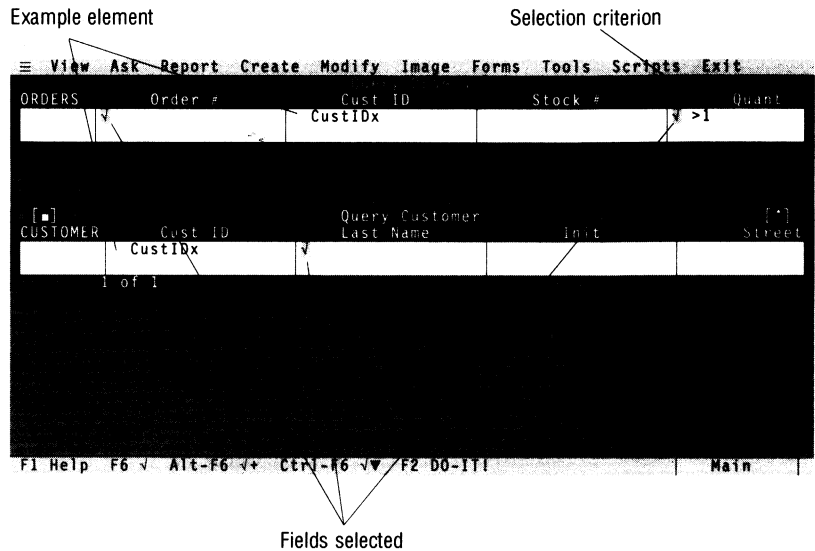
```

Query
Orders | Order # | Cust ID | Quant |
      | Check  | _CustIDx | Check >1 |

Customer | Cust ID | Last Name |
         | CustIDx | Check     |
Endquery

```

As you can see, the representation of a query form that is saved in a script closely matches what you see on the following Paradox screen:



Note these features:

- The query statement is enclosed by the keywords QUERY and ENDQUERY.
- Blank lines separate the QUERY and ENDQUERY keywords from the representation of the query forms.
- The structure of the query forms is reproduced using vertical lines.
- The keyword CHECK is used in the script to represent the Checkmark **F6**. (You can also use Check Descending **Ctrl-F6**, Check Plus **Alt-F6**, and Group By **Shift-F6** symbols in the query form.)
- Example elements are preceded by an underscore (\_).
- Selection criteria appear exactly as typed in the query form, as in >1.
- Only active fields in the query statement (those with checkmarks, examples, or selection criteria) are included.

## Abbreviated menu commands

Abbreviated menu commands are shortcuts for sequences of menu choices that perform many common Paradox operations. For

example, rather than using the script in Example 2-2 to sort the *Orders* table, you can use the following:

```
SORT "Orders" ON "Date" D ; sorts Orders by Date in descending order
```

The abbreviated menu command SORT is equivalent to choosing Modify from the Main menu, then Sort from the Modify menu, specifying a table, and filling out the sort form. You can test this equivalence by creating the two scripts and then playing both using Scripts | ShowPlay.

Abbreviated menu commands and their equivalent Paradox menu choices are listed in Table 2-2. These commands have the same syntax as PAL programming commands and can't be recorded directly from Paradox. Abbreviated menu commands require that Paradox be in the appropriate system mode in order to work. Main, Report, and Edit modes are examples of Paradox system modes. When you are using Paradox interactively, the system mode is shown in the lower right corner of the screen. You can also use the SYSMODE() function in a script to find the current mode.

Table 2-2 Abbreviated menu commands

Abbreviated command	System mode	Equivalent menu commands
ADD	Main	Tools More Add
CANCELEDIT	Edit, Form Designer, or Report Designer	Cancel Yes
COEDIT	Main	Modify CoEdit
COPY	Main	Tools Copy Table Replace
COPYFORM	Main	Tools Copy Form Replace
COPYREPORT	Main	Tools Copy Report Replace
CREATE	Main	Create
DELETE	Main	Tools Delete Table OK
DOS	All	Tools More ToDos
EDIT	Main	Modify Edit
EDITOR FIND	*	Search Find
EDITOR FINDNEXT	*	Search Next
EDITOR GOTO	*	Edit GoTo Line
EDITOR NEW	*	File New
EDITOR OPEN	*	File Open
EDITOR READ	*	File InsertFile
EDITOR REPLACE	*	Search ChangeToEnd
EDITOR WRITE	*	File WriteBlock

Abbreviated command	System mode	Equivalent menu commands
EMPTY	Main	Tools More Empty OK
EXIT	Main	Exit Yes
INDEX	Main	Tools Query Speed
LOCK	Main	Tools Net Lock
	Main	Tools Net PreventLock
MOVETO	Main	Image Zoom Record, Field, or Value
PICKFORM	Main, Edit, CoEdit	Image PickForm
PLAY	All	Scripts Play
PROTECT	Main	Tools More Protect Password Table
RENAME	Main	Tools Rename Table Replace
REPORT	Main	Report Output Printer
SETDIR	Main	Tools More Directory OK
SETPRIVDIR	Main	Tools Net SetPrivate
SETUSERNAME	Main	Tools Net UserName
SORT	Main	Modify Sort
SUBTRACT	Main	Tools More Subtract
UNDO	Edit, CoEdit	Undo Yes
UNLOCK	Main	Tools Net Lock
	Main	Tools Net PreventLock
VIEW	Main	View

\*Available during an Editor session in any mode

Though they provide no performance benefit when a script is executed, abbreviated menu commands have several advantages over equivalent braced menu choices:

- They are easier to type and easier to read.
- You can use variables or expressions as arguments. For example, to view a table whose name is currently the value of the variable *Tabname*, use the command

```
VIEW Tabname
```

This is not the same as

```
Menu {View} {Tabname}
```

which displays a table called *Tabname* (if there is one). Similarly, the abbreviated menu command

```
SORT TABLE() ON FIELD()
```

sorts the current table on the workspace by the values in the current field—something you couldn't specify with menu choices alone.

- They automatically provide confirmation when an action requires it. For example, the sequence

```
Menu {Tools} {Copy} {Table} {Orders} {Newords}
```

will copy the *Orders* table to *Newords* only if *Newords* doesn't already exist. If *Newords* does exist, Paradox displays a menu letting you either cancel the command or replace the table.

Thus, if your script guesses wrong about whether to include this {Replace} command, you get a script error. On the other hand, the abbreviated menu command

```
COPY "Orders" "Newords"
```

automatically provides confirmation if needed and will work whether or not *Newords* already exists.

(If you *didn't* want to replace a table inadvertently, you could use the `ISTABLE()` function to check whether it exists before you execute the `COPY` command. Or you could ask the user to provide confirmation as described in "Using recorded keypress interaction" in Chapter 18.)

Because of these advantages, as you refine an application, you may want to replace some or all menu choices you've recorded into scripts with abbreviated menu commands. Since they have exactly the same effect as the menu choices they reproduce, abbreviated menu commands can be used only in contexts in which they make sense. For example, you'll get an error if you use the command

```
VIEW "Orders"
```

while in Edit mode or in some other context in which View is not a valid menu choice. It is your responsibility as script writer to ensure that abbreviated menu commands are used when Paradox is in the proper mode (use the `SYSMODE()` function to determine the current mode). You can use the `TYPEIN` and `SELECT` commands to build your own abbreviated menu commands for menu choices other than the ones in Table 2-2. For details, see "Simulating keypress interaction" in Chapter 18.



# Expressions

Arguments to PAL commands or functions are called *expressions*. For example, in this MESSAGE command

```
MESSAGE "Tomorrow's date is ", TODAY() + 1
```

there are two arguments (expressions), "Tomorrow's date is " and TODAY() + 1.

Expressions are built from

- data values (constants)
- variables and array elements
- operators
- field specifiers
- function and procedure calls
- fixed and dynamic arrays

This chapter describes each of these elements and how it is used. It also explains

- how expressions are typed and evaluated
- how to manipulate strings, dates, and keycodes in expressions

---

## What is an expression?

An expression is a single value, or a set of one or more elements that evaluate to (or result in) a value. Values are data that have one of the PAL data types: alphanumeric, numeric, currency, short number, date, or logical. Since commands often perform actions on values, you'll find that many PAL commands and functions require an expression as an argument or parameter.

Here are some examples of simple expressions:

```
5                ; the number 5 (type = number)
"abracadabra"   ; the string "abracadabra" (alphanumeric)
"Hi, " + "there" ; the string "Hi, there" (alphanumeric)
PI() * (r * r)   ; the value calculated by squaring the value
                ; of the variable r, then multiplying the
                ; result by the value returned by the
                ; function PI()
GETCHAR()       ; the ASCII code of the next character
                ; typed at the keyboard (number)
TODAY() < 12/31/92 ; True or False, depending on
                ; today's date (logical)
```

The first expression, 5, is an example of a numeric value. The second is a literal string. The third concatenates (combines) two string values. The fourth uses arithmetic operators to manipulate values stored in variables. The fifth is a function, while the last is a comparison between a function and a date value. Expressions can be any of these elements by themselves, or combinations of elements. Expressions can be up to 132 characters in length.

---

## Data types

PAL expressions evaluate to one of seven data types:

- alphanumeric (**A**) or (**M**); strings of any length from 0 to approximately 64 million characters
- number (**N**); range of approximately  $\pm 10^{-307}$  to  $\pm 10^{308}$
- currency (**\$**); same range as number
- short number (**S**); range of -32767 to 32768
- date (**D**); any valid date between 1/1/100 and 12/31/9999
- logical (**L**); True or False
- fixed array (**AY**) or dynamic array (**DY**)
- procedure; a modular set of PAL statements that perform a specific task

The first four data types are the standard Paradox field types. They are discussed in detail in Chapter 2 of the *User's Guide*. In PAL, a string of 255 or fewer characters corresponds to the alphanumeric (A) field type; longer strings correspond to the memo (M) field type. A string is transparently converted from type (A) to type (M), depending on its length. Strings up to 255 characters in length are kept in memory and allow fast access; up to 64K of the portion greater than 255 characters may also be kept in memory for fast access.

The last three data types are available only in PAL scripts. They cannot be used as field types in a table.

Logical values are either True or False (sometimes they are referred to as *Boolean*). For example, the expression

```
TODAY() < 12/31/90
```

returns the value of True if the current date is before the end of 1990, and False otherwise. Although the data type logical is available only in PAL scripts, you can obtain the effect of a logical field in a table by using PAL pictures. See the discussion of pictures in Chapter 5.

The fixed and dynamic array data types are versatile storage for sets of values. They are discussed in detail later in this chapter.

Procedures are a special data type. Most expressions are restricted from using procedures as a data type. However, the *ProcName* in the statement

```
RELEASE PROCS ProcName
```

is an expression that evaluates to the name of a procedure.

Subject to some limitations, you can mix data values of different types in expressions, and PAL automatically assigns a data type to the resulting expression. The limitations and rules used to determine the type of a mixed expression are shown in Table 3-1.

Table 3-1 Resulting data type of mixed expressions

Start with...	Mix with...					
	N (number)	A (alphanumeric)	D (date)	\$ (currency)	S (short number)	L (logical)
N (number)	N	*	*	\$	N	*
A (alphanumeric)	*	A	*	*	*	*
D (date)	D	*	N	*	D	*
\$ (currency)	\$	*	*	\$	\$	*
S (short number)	N	*	*	\$	S	*
L (logical)	*	*	*	*	*	L

\* not allowed

## Evaluating expressions

You can evaluate an expression at any time by pressing PAL Menu *Alt-F10* and choosing Value. Value, which also serves as a general calculator, lets you interrogate the values of variables, array elements, or any other expression during debugging, or from any place in Paradox. For more information, see the description of Value in Chapter 8.

The order in which elements of an expression are evaluated is discussed under "Operators" later in this chapter.

---

## Elements of expressions

PAL expressions can include the following kinds of elements:

- constants
- variables
- fixed and dynamic arrays
- operators
- field specifiers
- functions (either built-in functions or user-defined procedures)

---

### Constants

The simplest expression elements are alphanumeric, numeric, date, and logical *constants*. Constants are so named because their value does not change—a constant's value remains the same throughout its lifetime in a script.

---

#### Alphanumeric constants

Alphanumeric constants are also called *strings* because they consist of a string of ASCII characters. In PAL commands, string constants are always enclosed in double quotation marks ("" ) to distinguish them from other kinds of expressions. Strings can be from 0 to approximately 64 megabytes in length. A string constant can span more than one line by concatenation with the + operator. Alphanumeric constants correspond to two data types: "A" for alphanumeric and "M" for memo.

You can run the following examples with the MiniScript command from the PAL menu *Alt-F10*:

```
RETURN "Hello"  
  
RETURN " this one contains punctuation,..! and spaces at the ends "  
  
RETURN "99.994"  
  
RETURN "1/31/90"
```

Although the last two examples look like numeric and date values, they are strings because they are in quotes.

See the discussion of the STRVAL() and NUMVAL() functions in the *PAL Reference* for details on turning strings into numbers and vice versa.

---

#### Special characters

Some ASCII characters cannot easily be typed directly into an alphanumeric constant for one of two reasons:

- They have no corresponding key (like the Greek letter  $\alpha$ ).

- The corresponding key has a special meaning in a script (like a double quotation mark).

There are two ways to include such characters in a constant. Both use ASCII character codes (see the *PAL Reference* for a complete list).

- Hold down *Alt* while typing the character's numeric ASCII code on the *numeric keypad* (do not use the numbers above the main alphabetic keys). For example, to include  $\alpha$  in a constant, press *Alt-224* on the numeric keypad.
- Use a backslash (\) to precede either the character itself or the character's numeric ASCII code.

Here are some sample program lines using backslash sequences:

```
RETURN "Quote mark coming\''"      ; text ends with one quotation mark
RETURN "Backslash away\\"          ; text ends with one backslash
RETURN "Give me a Ctrl-R:\018"    ; text ends with Ctrl-R character
RETURN "\224 to \234"             ;  $\alpha$  to  $\omega$  (alpha to omega)
RETURN "\012"                     ; ASCII formfeed character
```

Certain frequently used ASCII codes, such as the tab character (ASCII 9 or *Ctrl-I*), have special backslash sequences similar to those in the C programming language. These sequences, which are shown in Table 3-2, are preferred in strings because they are more readable than their numeric equivalents. They are especially useful for controlling a printer (in a PRINT statement).

Any character not shown in Table 3-2 that follows a backslash in a string is taken literally—the backslash is ignored. For example, “\a” is just the same as “a”.

Table 3-2 Backslash sequences

Backslash sequence	Function	Equivalent keystrokes
\t	Tab	<i>Ctrl-I</i> or <i>Tab</i>
\n	Newline	<i>Ctrl-J</i> or <i>Enter</i>
\f	Formfeed	<i>Ctrl-L</i>
\r	Carriage return	<i>Ctrl-M</i>
\"	Quotation mark	"
\\	Backslash	\

### **Numeric constants**

Numeric constants are written as a sequence of digits, optionally preceded by a minus sign (–) and optionally containing a decimal point.

As an alternative to typing a number literally, you can use scientific notation to represent numbers, which is especially convenient for very large and very small numbers. Numbers in scientific notation begin with the decimal value and end with the letter *E* followed by the exponent, which can be signed (+ or -).

Here are some examples of numeric constants:

```
x = 25 RETURN x           ; the number 25
x = 3.1514 RETURN x       ; a number with a decimal point
x = -17.00000001 RETURN x ; a negative value with a small fraction
x = 5.6E+9 RETURN x       ; 5.6 * 109, or 5,600,000,000
```

When writing numeric constants, note that they cannot contain dollar signs or whole number separators, and enclosing them in parentheses does not make them negative.

Numeric constants are stored internally as floating-point numbers ranging from  $\pm 10^{-307}$  to  $\pm 10^{308}$ . This specifies a number with approximately 15 digits of precision. Numeric constants may be used for numeric (N), currency (\$), or short number (S) values.

See the discussion of the STRVAL() and NUMVAL() functions in the *PAL Reference* for details on turning strings into numbers and vice versa.

---

***Numeric constants and the international number format***

If a non-U.S. country group other than U.K. was specified when Paradox was installed, numbers and currency will be displayed using the international number format (for example "One thousand forty-six and seven tenths" is represented as 1.046,7). The display format for numbers can also be changed using the Custom Configuration Program (see Chapter 15 of *Getting Started*).

If you are using the international display format for numbers and currency (not to be confused with an international sort order), all PAL operations respect this format with one exception: entering numeric constants in expressions. If you want to include a numeric constant in an expression, such as when you assign a value to a variable, you must use the U.S. format for numbers; that is, you must use a point (.) and not a comma (,) to indicate the decimal portion of a number. Thus, the following expressions

```
y = 2.1
x = y + 3.2
```

are correct even though the international number convention is in effect.

The use of numeric constants in PAL with the international number convention in effect is shown in the following examples:

```

x = 23.5           ; value must be assigned using U.S. convention
@1,1 ?? x         ; show value of x, displayed as 23,5
y = 2.4           ; again, constant must be entered in U.S.
                  ; format
@2,2 ?? x + y     ; displayed as 25,9

z = STRVAL(25.9)  ; numeric parameter passed to function must
                  ; be entered in U.S. format
                  ; (true for user-defined procedures as well)
@ 4,4 ?? z        ; will be "25,9"

q = NUMVAL("25,9") ; string representation of a number is shown
                  ; in international format
NUMVAL("25,9") = NUMVAL(STRVAL(25.9)) ; is True

```

Paradox lets you use many of the PAL functions in calculated fields in forms and reports. If you are using the international number format, you need to differentiate between the comma used as an argument separator in a function that takes more than one argument and a comma as a decimal point within a single value. To do so, put a space before and after a comma when you're using it as an argument separator in a calculated field.

---

### **Date constants**

Date constants can be represented in either mm/dd/yyyy, dd-Mon-yyyy, dd.mm.yyyy, or yy.mm.dd format. For dates in the twentieth century, the "19" in the year may be omitted. Thus the following dates are equivalent:

```

12/17/1989
12/17/89
17-Dec-1989
17.12.89

```

Correspondingly, dates in the second through the tenth centuries must include three digits of the year (as in 12/17/243); dates in the eleventh through nineteenth centuries, and the twenty-first century and beyond, must have four digits (12/17/1043). The year cannot be omitted completely.

A date constant must represent a valid date. Paradox knows which months have 30 and 31 days and in which years February has 29. Paradox also knows about leap centuries, should your dates range that far.

Note that dates cannot contain spaces around the slashes. For example, Paradox treats the following expression as a date:

```
x = 7/17/1956
```

However, Paradox treats the following expression as a numeric data type:

```
x = 7 / 17 / 1956
```

because the slashes are interpreted as a sequence of division operations.

See “Field types” in Chapter 2 of the *User’s Guide* for complete information about handling dates in Paradox.

---

### **Logical constants**

There are two logical constants:

True  
False

You can spell them in any mixture of upper- and lowercase letters.

Use the `FORMAT()` function (see Chapter 5 in this book and the *PAL Reference*) to change the output of logical expressions to On/Off or Yes/No instead of True/False.

---

### **Blank values**

In Paradox, the value of any field in which no data is present is blank. The value of any PAL expression (except those that evaluate to type logical) can also be blank.

A blank string value (memo or alphanumeric) is denoted by the empty string (“”) and not a space (ASCII 32). Blank numbers and dates can be generated by the `BLANKNUM()` and `BLANKDATE()` functions respectively. You can use the `ISBLANK()` function to test whether the value of an expression is blank.

If you want to indicate a value of zero in a numeric field of a Paradox table, it is always preferable to explicitly enter the value rather than to leave the field blank.

There is a special `Blank=Zero` option in the Custom Configuration Program that you can use to treat blanks as zeros in calculations. Unless this option has been enabled, any PAL calculations involving blank numeric values result in a script error. In a numeric field on which you expect to perform calculations, you should require that the field takes either a value or zero. You can use the `ISBLANKZERO()` function to check on the current settings.

---

### **Variables**

A *variable* is like a slot in which a script can temporarily store one item of information for use elsewhere.

Variables are used for temporary storage of values. The values of variables can be any PAL data type. You can use the `TYPE()` function to determine the data type of the value of a variable. For example,

```
x = 35  
RETURN TYPE (x)
```

returns the value *N* to indicate that the value of *x* is a number. See the *PAL Reference* for a complete description of the `TYPE()` function.

---

### **Naming conventions**

Here are the rules for naming the PAL variables, arrays, and procedures you use in scripts:



- Names can be up to 132 characters in length.
- The first character must be a letter *A – Z* or *a – z*.
- Subsequent characters can be letters, digits, or one of these characters: *. \$ ! \_*
- No spaces or tabs are allowed.
- Distinctions between upper- and lowercase letters are ignored.
- Names cannot duplicate Paradox reserved words. (See Appendix C in the *PAL Reference* for a complete list of reserved words.)

**Table 3-3 Valid and invalid naming conventions**

<b>Name</b>	<b>Valid?</b>	<b>Explanation</b>
Anything	yes	
time.period	yes	
!claim	no	doesn't begin with a letter
claim!	yes	begins with a letter
L0123	yes	
5A6	no	doesn't begin with a letter
abc xyz	no	contains a space
abc_xyz	yes	
menu	no	reserved word

These rules govern only the names of PAL variables, arrays, and procedures. See Chapter 2 of the *User's Guide* for naming rules for Paradox objects—tables, fields, forms, reports, graphs, and scripts.

Although the names of PAL elements can duplicate the names of Paradox objects, it is not recommended. Suppose you've used *Orders* both as a table and as a variable storing the name of the last customer to place an order. Now you want to create a new table with that customer's name. If you inadvertently include double quotation marks and type

```
CREATE "Orders"
```

your script creates a table with the name *Orders*—replacing the one you've already got. Similarly, if you want to view the *Orders* table but type

```
VIEW Orders
```

you'll get the table with the customer's name (if one exists). Debugging will be easier if you make all names unique.

Similarly, you can use a reserved word within a valid PAL name, such as a variable called *IsEdit* whose name contains the menu

command Edit. However, this makes it easy to confuse elements you create (such as variables and arrays) with valid PAL commands. If you're not familiar with the set of PAL commands and functions, for example, you might think that *IsEdit* is a PAL function that tells you whether Paradox is in Edit mode. It's safer to create unique names that cannot be confused with the names of PAL and Paradox commands and actions.

---

## Assigning values to variables

To create a variable, assign it a value—no explicit declaration is necessary. The most common way of giving a variable a value is to use the assignment command (=). For example,

```
x = "abc"
```

assigns the string value "abc" to variable *x*. If *x* had been assigned a value previously, the former value would now be lost.

Certain other commands (such as FOR, SHOWMENU, ACCEPT, and LOCATE) also assign values to variables as side effects to the action of the command.

You must assign values to variables before you use them in expressions. If, for example, the command

```
MESSAGE "The value of x is ", x
```

is executed before *x* is assigned a value, you'll get a script error. You can use the ISASSIGNED() function to test whether a variable has been assigned a value. Variables of any type other than logical can be assigned a blank value (see "Blank values" earlier in this chapter).

You do not need to explicitly indicate a data type for PAL variables. The data type of a variable is simply the type of the value it currently holds. For example, the sequence of commands

```
x = "abc"  
x = 5 + 5  
x = 16-Dec-1989
```

makes *x* first an alphanumeric variable, then a number, then a date.

---

## Using variables

To insert the value of a variable in an expression, type the variable name. Here are some examples:

```
x = 16-Dec-1989  
MESSAGE "The value of x is ", x ; displays value of variable x  
  
MOVETO [Date Hired] ; moves to Date Hired field and  
TYPEIN x ; enters value of variable x  
  
IF NOT (ISASSIGNED(rank)) ; assigns 0 to variable rank if  
THEN rank = 0 ; currently unassigned  
ENDIF
```

To use a variable in a query statement, precede its name with a tilde (~) as shown in Example 3-1.

Outside of the context of procedures, PAL variables are always *global*—they retain their values for the duration of the Paradox session in which they are used. For example, you can assign a value to a variable in one script and use it later from a different script. The value is erased only when you

- exit from Paradox
- assign a different value
- use a RELEASE VARS command to explicitly erase the variable

Within procedures, the rules for determining the scope of a variable are somewhat more elaborate. See “Variables and procedures” in Chapter 6.

### Example 3-1 Using tilde variables in queries

This script looks up the customer in the *Customer* table whose name is stored in the variable *name*. If it's there, it displays the table; if not, it displays a message to that effect.

```

QUERY
    Customer | Cust ID | Last Name | Init | Street | City |
             | Check   | Check ~name | Check | Check  | Check |
ENDQUERY
DO_IT!                ; execute query

IF (ISEMPTY("Answer")) ; if no records were found
    THEN MESSAGE name, " was not found."
        SLEEP 2000
        MESSAGE ""
    ELSE
        WAIT TABLE ; let user examine table
        PROMPT "Press [F2] when done."
        UNTIL "F2"
ENDIF

```

Tilde variable used to modify query at run time

You can use the SAVEVARS command (see the *PAL Reference*) to save the current values of certain (or all) variables into a special script file called *Savevars*. Once saved, you can restore the variables by playing that script later in the current session or in a subsequent one. This feature is useful for debugging, since it lets you take a snapshot of the current state of variables.

It is good programming practice to release all variables with the RELEASE VARS ALL command at the end of an application or top-level script play before returning control to Paradox. This frees the memory they occupy and ensures that values will not accidentally be passed to another script played later. Of course, if you

intend to pass values to another script that might be played later in the same Paradox session, you don't want to release the variables. If you don't, however, be aware that a user could modify the variables by choosing MiniScript from the PAL menu or by playing an intervening script that uses the same variable names.

**Note** If you are using procedures, private variables are automatically released from memory when the procedure terminates. In addition, when closed procedures terminate, they automatically release all variables declared within their scope. See Chapter 6 for details.

---

### System variables

Three system variables have special meaning in PAL:

- ❑ *Autolib*, the autoloading library path (see "Autoloading procedures" in Chapter 6)
- ❑ *Errorproc*, an error-handling procedure (see "Error processing" in Chapter 7)
- ❑ *Retval*, a returned value or key (see "Using Retval" next)

---

### Using Retval

These commands, among others, cause PAL to assign a value to the system variable *Retval*:

- ❑ When the RETURN command is used to return a value to a calling script or to Paradox, *Retval* is assigned the returned value.
- ❑ When ACCEPT is used to input a value, *Retval* is set to a logical value of True if the user enters a value, otherwise, *Retval* is set to false.
- ❑ When LOCATE is used to search for a record, *Retval* is set to a logical value of True if a matching record is found, otherwise, *Retval* is set to false.
- ❑ When LOCK, LOCKKEY, LOCKRECORD, UNLOCK, or UNLOCKRECORD is used to lock or unlock a table or record, *Retval* is set to a logical value of True if the operation was successful, and to False if not. For example, *Retval* will be false if LOCK tries to lock a table that's already locked, or if UNLOCK tries to unlock a table that isn't locked.
- ❑ When the user ends a keypress-driven WAIT interaction, *Retval* is assigned the keycode of the key that caused the WAIT to end. When an event-driven WAIT terminates, *Retval* is assigned a special value to indicate the specific situation that caused the WAIT to end.

You will find *Retval* handy in various circumstances. For example, you can test *Retval* in multiuser applications after trying to lock a table to find out whether the lock was successful and if the table is

therefore available for use. The *PAL Reference* describes the effect that commands have on the system variable *Retval*.

When mixing *Retval* with other values in expressions, remember that it automatically changes type according to the value returned in the context in which it is set.

---

## Fixed and dynamic arrays

Arrays are used for the temporary storage of a set of data values. PAL supports two kinds of arrays: fixed arrays and dynamic arrays. Fixed arrays have a set number of elements; each element is accessed by a number that specifies the location of that element in the array. Dynamic arrays have a variable number of elements. Each element of a dynamic array is accessed by its unique tag or label. PAL arrays are *single-dimensional*. This means you can't declare an array of arrays.

---

### Fixed arrays

Unlike variables, you must declare fixed arrays before you can store values in them (the exceptions are arrays created by the COPYTOARRAY command, described in Chapter 19). You do this with the ARRAY command, which allocates space in memory for the size array you specify. You can use any kind of numeric expression in the ARRAY command. For example, both

```
ARRAY a[5]
```

and

```
x = 2  
ARRAY a[x + 3]
```

declare an array named *a* consisting of five data elements:

```
a[1] a[2] a[3] a[4] a[5]
```

You don't have to declare a data type for the elements of an array; in fact, the elements can be different data types.

As with variables, storage for arrays is allocated automatically by Paradox's virtual memory management system. You can declare as many arrays as you want, each with up to 15,000 elements.

---

### Fixed array elements

Array elements are referenced by numeric subscripts enclosed in square brackets; for example, *a*[2] refers to the second element of array *a*. The subscript provides an *index* to the array. You can think of each element of an array as a separate variable because they are assigned and used in the same way as variables. Like variables, array elements must first be assigned values before they can be used in expressions:

```
ARRAY b[5]           ; declare b as an array of five elements  
x = 2                ; assign x the value of 2  
b[x + 1] = 10        ; assign element b[3] the value of 10
```

Then:

```
RETURN b[3] + 2      ; returns the value 12
RETURN b[4] + 2      ; results in a run error since b[4] is unassigned
```

In the expression  $b[x + 1]$ ,  $x + 1$  is called a *subscript expression*. Subscript expressions usually must evaluate to an integer between 1 and the size of the array they are used to access. Under special circumstances, they may evaluate to the name of a field of a Paradox table (see the discussion of COPYTOARRAY and COPYFROMARRAY in Chapter 19 for details).

Unlike most programming languages, PAL lets you store values of different data types in elements of the same array. For example, the following sequence of commands is valid:

```
ARRAY b[3]          ; declare b as an array of three elements
b[1] = "Abc"        ; assign element b[1] an alphanumeric value
b[2] = 3.1415       ; assign element b[2] a number value
b[3] = 12/17/89     ; assign element b[3] a date value
```

As with variables, reassigning an element erases its previous value and can change its data type.

The ARRAYSIZE() function can be used to determine the size of an existing array. For instance, if  $b$  is an array of three elements, returning the value of ARRAYSIZE( $b$ ) displays 3.

You can use the TYPE() function to determine if the data type of a variable is a fixed array. For example, if  $b$  is an array of three elements, the statement TYPE( $b$ ) displays AY. You can also check the type of individual array elements; for example, the statement TYPE( $b[2]$ ) displays N. See the *PAL Reference* for a complete description of the TYPE() function.

---

### ***Lifetime of arrays and array elements***

Like variables, arrays and their elements retain their values for the duration of the Paradox session in which they are used. Use the RELEASE VARS command to undeclare an array. If an array is released or redeclared, its previous declaration—and the values of all of its elements—are forgotten. For example, the script

```
ARRAY a[20] ; declares the array a with 20 elements
a[5] = 6
a = 42      ; redeclares the variable a
? a[5]
```

will cause a script error because the array was released when the variable  $a$  was redeclared as a numeric variable type.

As with variables, the rules are somewhat more elaborate when arrays are used in the context of a procedure. See Chapter 6 for details.

---

### Using arrays with records

Arrays are useful for temporarily storing an entire record from a Paradox table. For example, you can use arrays to edit a record before saving its contents or to move a record from one table into another.

The special PAL commands COPYTOARRAY, COPYFROMARRAY, and APPENDARRAY make it easy to work with records in arrays. When you use them, you can refer to array elements by their field names in place of their subscript positions.

---

### Dynamic arrays

Dynamic arrays are similar to fixed arrays:

- Although dynamic arrays must (usually) be declared, you do not declare a specific size for a dynamic array. Dynamic arrays are sized dynamically—they grow and shrink as needed. Dynamic arrays can grow beyond the fixed array limit of 15,000 elements.
- The elements of a dynamic array ARE accessed without using a subscript. Instead, you use something called a *tag*. A tag is nothing more than a label for each element in the array.

A dynamic array is a compact storage device for any combination of data types. Because dynamic arrays use associative memory, PAL lets you look up values quickly, even when the dynamic array contains a large number of elements.

---

### Declaring dynamic arrays

You must declare a dynamic array before you can store values in it, unless you create one “on the fly” by using a command such as WINDOW GETATTRIBUTES, GETCOLORS, or SYSINFO. For example, you would declare a dynamic array called *GroceryBag* as follows:

```
DYNARRAY GroceryBag[]
```

---

### Dynamic array elements

To assign values to the elements of the dynamic array *GroceryBag*, you could use this sequence of commands:

```
Tag ----- GroceryBag["Type"] = "Paper"  
GroceryBag["Size"] = "Large"  
GroceryBag["Double"] = True  
GroceryBag["Fruit"] = "Apples"  
GroceryBag["Soda"] = "Root Beer"  
GroceryBag["Total"] = 3.29  
GroceryBag["Frozen"] = False  
GroceryBag["Date"] = Today()
```

The elements within dynamic arrays are not ordered sequentially (as fixed array elements are). Dynamic arrays use associative memory to evaluate an element: The value of an element in a dynamic array is retrieved by referencing its tag, rather than its position within the array.

For example, the statement

```
RETURN GroceryBag["Size"]
```

displays the string *Large* as the value of the element *Size*.

Tag names can be integers, floating-point numbers, text strings up to 255 characters in length, dates, or any expression that evaluates to an integer, floating-point number, string, or date.

You can determine the size of a dynamic array with the `DYNARRAYSIZE()` function. The statement

```
RETURN DYNARRAYSIZE(GroceryBag)
```

displays the value 8.

You can determine that the variable *GroceryBag* is a dynamic array (DY) with the statement

```
RETURN TYPE(GroceryBag)
```

See the discussion of `FOREACH` in Chapter 4 for details on `FOR` loops and dynamic arrays.

---

## Operators

You can use operators in expressions to combine and manipulate values and data elements. Table 3-4 is a complete list of operators available in PAL.

Table 3-4 PAL operators

Operator	Function
<b>Alphanumeric (A)</b>	
+	concatenation
<b>Numeric (N, \$, S)</b>	
+	addition
-	subtraction or negation
*	multiplication
/	division
<b>Date (D)</b>	
+	addition
-	subtraction or days apart
<b>Comparison</b>	
=	equal to
<>	not equal to
<	less than



Operator	Function
<=	less than or equal to
>	greater than
>=	greater than or equal to
<b>Logical</b>	
AND	logical AND
OR	logical OR
NOT	logical NOT

In the examples in this section, assume that these variables have been assigned:

```
Age = 50.2
Deathly = " Tax"
NewYear = 1-Jan-1990
```

### The + operator

The action of the + operator depends upon the type of expression it's operating on:

- number + number* adds the two numbers
- string + string* combines the strings (called *concatenation*); string concatenation is legal on alphanumeric fields as well as memo fields
- date + number* adds a number of days to the date (called *date addition*)

```
RETURN 45 + 50.2 + 5 ; 100.2 (addition)
RETURN 45 + Age + 5 ; 100.2 (addition)
RETURN "Income" + " Tax" ; "Income Tax" (concatenation)
RETURN "Income" + Deathly ; "Income Tax" (concatenation)
RETURN 12/17/89 + 7 ; 12/24/89 (date addition)
RETURN NewYear + 1 ; 2-Jan-90 (date addition)
```

No other orderings of arguments in expressions combining different data types are permitted with the + operator. For instance, the fifth example could not be written `7 + 12/17/89` since *number + date* is not allowed, only the reverse.

### The - operator

Like the + operator, the - operator has multiple uses:

- number - number* subtracts the second number from the first (called *subtraction*)
- number* reverses the sign of the number (called *unary negation*)
- date - number* subtracts a number of days from a date (called *date subtraction*)
- date - date* calculates the number of days between two dates (called *days apart*)

```

RETURN 76.5 - Age           ; 26.3 (subtraction)
RETURN -Age                 ; -50.2 (negation)
RETURN NewYear - 3         ; 28-Dec-1989 (date subtraction)
RETURN TODAY() - NewYear   ; the number of days since
                           ; Jan. 1, 1990 (days apart)

```

No other combinations of argument types are permitted with the `-` operator. In particular, you can't use `-` with string arguments. (However, there are string functions you can use to extract substrings; see "String manipulation" later in this chapter for details.)

---

## ***The \* and / operators***

The multiplication (`*`) and division (`/`) operators can be used only with numeric arguments. Division by zero does not result in an error, but results in a value of zero.

To trap attempted division by zero as an error, you can construct a simple routine. For example,

```

; trap for division by zero

WINDOW CREATE @3,5 WIDTH 35 HEIGHT 5 TO InputWindow
@I,1 ?? "Enter number to be divided: "
ECHO NORMAL
ACCEPT "N" TO x
@I,1 ?? "Enter number to divide by : "
ACCEPT "N" TO y

IF y <> 0
    THEN z = x / y
    ELSE z = "Error"
ENDIF

RETURN z

```

You can also create a procedure for this error condition as part of a library of math rules. For example,

```

PROC Division(x,y)
    IF x = "Error"
        OR y = "Error"
        OR y = 0
        THEN RETURN "Error"
        ELSE RETURN x/y
    ENDIF
ENDPROC

```

See Chapter 6 for a complete description of procedures.

---

## ***Comparison operators***

You can use the comparison operators in Table 3-4 to compare values of any of the data types, including logical. The result is always a logical value, True or False. The action of operators like `<` and `>` depends on the types of arguments being evaluated.

Table 3-5 Comparison rules

Data type	Basis	Ordering
Numeric	Numeric Order	Lower < Higher
Alphanumeric	Alphanumeric order	Depends on sort order
Date	Chronological order	Earlier < Later
Logical		False < True

Alphanumeric comparisons depend on the sort order established by the current PARADOX.SOR file. In the ASCII sort sequence,  $A < Z < a < z$ , but this is not true in the other sequences, which support a true dictionary sorting order (that is, characters are sorted in alphabetic sequence irrespective of case). For details on the available sort orders, see Chapter 22 of the *User's Guide*. Sort order is established at installation, but can be changed later (see Chapter 15 of *Getting Started* for details).

Blank values are considered to be less than all other values of the same data type.

You can compare alphanumeric (A) fields with memo (M) fields. However, if you compare other values of different types,  $<>$  returns a value of True, while all other operators return False. For example:

```
RETURN 5 + 1 < 6 * 2      ; True (see also "Order of evaluation")
RETURN Age > 21           ; True
RETURN 7/4/1776 = NewYear ; False, since the dates are not the same
RETURN "Abc" = "ABC"     ; False, since the strings are not the same
RETURN "Abc" > "ABC"     ; True in ASCII order, false in other orders
RETURN Age > Age + 5     ; False
RETURN 3 = "ABC"         ; False, since the types are mixed
RETURN 3 <> "ABC"        ; True, since the types are mixed
RETURN "1" > FILL("1",300) ; False, since the alpha type is smaller than
                        ; the memo type
```

Note that the = sign is used both as a comparison operator within expressions and as part of the assignment command. When only a variable name appears to the left of an = sign, it is assigned the value of the expression to the right; otherwise, the two expressions are compared. So,

```
a = 3      ; assigns the value of 3 to a
x = 4 = a  ; the first = is an assignment, the second a comparison
```

Here, the last statement assigns to the variable x the value of the expression  $4 = a$  (False). Enclose expressions in parentheses when there is any chance of confusion:

```
x = (4 = a)
```

### Logical operators (AND, OR, NOT)

Expressions with logical values can be combined using the logical operators in Table 3-4:

- $x$  AND  $y$  returns True if both arguments ( $x$  and  $y$ ) are true.
- $x$  OR  $y$  returns True if either argument (or both) is true.
- NOT  $x$  returns True if the argument is false.

For example,

```
RETURN 3 = 4 AND 7 = 8      ; False, since neither argument is true
RETURN 3 = 4 OR 7 = 8      ; False, since both arguments are false
RETURN 3 = 3 OR 7 = 8      ; True, one argument is true
RETURN NOT (3 = 4 AND 7 = 8) ; True, since argument is false
RETURN NOT False           ; True, since argument is false
RETURN 3 <> 4              ; True, since the arguments are unequal.
```

**Note** There is a difference between the comparison operator  $<>$  (not equal to) and the logical operator NOT. The  $<>$  compares any two data values while the NOT operator negates a logical value.

---

### Order of evaluation

In expressions containing more than one operator, the operations are evaluated in the order of precedence shown in Table 3-6. Any expression contained in parentheses is evaluated first, and inner levels of parentheses are evaluated before outer levels. Where there are two or more operators of equal precedence in a single expression, they are evaluated from left to right.

To illustrate:

```
3 + 4 * 5          ; returns 23 (* over +)
(3 + 4) * 5        ; returns 35
((3 + 4) * 5) / 2  ; returns 17.5 (inner parentheses first)
```

Table 3-6 Order of precedence of PAL operations within expressions

Precedence	Operators
1	()
2	* /
3	+ -
4	= <> < <= > >=
5	NOT
6	AND
7	OR

---

### Field specifiers

PAL field specifiers make it easy to reference and make use of the values of fields in images on the Paradox workspace. All field specifiers are enclosed in brackets [ ] as shown in Table 3-7. They always point to the current record in an image.

You can use field specifiers to refer to fields in both display and query images. It makes no difference whether a display image (that is, the image of a table) is in table or form view.

In the current image, the current field is the one the cursor is in. In other images, the current field is the one the cursor was in most recently.

When you use a field specifier in an expression, it normally stands for the value currently entered in that field. (An exception is the MOVETO command, which is discussed in "Moving to specific fields" later in this chapter.) In a display image, the value will have the same data type as the field. In a query image, the value will be a string representation of the field's contents.

Table 3-7 Field specifiers

Format	Meaning*
[ ]	Current field in current image
[#]	Current record number in current image
[field]	Named field in current image
[table -> ]	Current field in the named table
[table -> field ]	Named field in the named table
[table(n) -> ]	Current field in nth image in named table
[table(n) -> field ]	Named field in nth image of named table
[table(Q) -> ]	Current field in query image of named table
[table(Q) -> field ]	Named field in query image of named table

\*Where table is a valid table name; field is a valid field name in the table; and n is the nth display image of a table on the workspace

### Using field specifiers in expressions

You can use field specifiers in expressions exactly as you would constants or variables. For example,

```
x = 3 + [ ]           ; adds 3 to value of current field and
                    ; assigns sum to x
IF ([Date] >= 1/1/90) ; tests whether value in Date field in
                    ; current record is January 1, 1990 or later
[ ] = x              ; assigns value of x to current field

[Orders->Quantity=10] ; assigns value 10 to the Quantity field
                    ; in the Orders table
[Total]=[Quant]*[Price] ; assigns the product of the Quant and Price
                    ; fields to the Total field
```

As this example shows, you can modify the value of a field in an image by specifying the field on the left side of an assignment statement. For instance,

```
[Quantity] = [Quantity] + 1
```

increments by 1 the value of the Quantity field in the current record of the current image.

### Example 3-2 Referring to fields

Suppose the workspace looks like this, with the cursor in the City field of the Customer table. The labels show which field each field specifier refers to.

[Customer -> Last Name]
[ ] or [Customer -> ] because cursor is on this record

The screenshot shows a workspace with three overlapping tables:

- Customer Table:** Columns: Cust ID, Last Name, Init, Street, City. Data rows: (5720, Helms, D, 52 Brattle Street, Cambridge), (5855, Chin, F, Hotel Orient, Jurong), (6125, Hawes-Anderson, D, Waves Cottage, Palm Springs), (6666, Matthews, J, 1050 12th Street, San Francisco).
- Orders Table (top):** Columns: ORDERS, Order #, Cust ID, Stock #, Quant. Data rows: (1, 2280, 4277, 130, 1), (2, 3351, 3266, 519, 1).
- Orders Table (bottom):** Columns: ORDERS, Order #, Cust ID, Stock #, Quant. Data rows: (1, 2280, 4277, 130, 1), (2, 3351, 3266, 519, 1), (3, 8070, 6125, 632, 8), (4, 6235, 2779, 890, 1).
- Query Orders Table:** Columns: ORDERS, Order #, Cust ID, Stock #, Quant. Data row: (find, 9004, , , ).

Labels and arrows in the image indicate field specifiers:

- [Orders (1) -> Stock#] points to the Stock # field in the top Orders table.
- [Orders (2) -> Cust ID] points to the Cust ID field in the bottom Orders table.
- [Orders (Q) -> #] points to the Order # field in the Query Orders table.
- [Orders (Q) -> Cust ID] points to the Cust ID field in the Query Orders table.

[Orders (Q) -> #]                      [Orders (Q) -> Cust ID]

To try any of these field specifiers, press **Alt-F10** to display the PAL menu, choose Value, then type in a field specifier. For example,

[Orders (Q) -> #]  
displays Find in the message window.

When there is more than one image of a particular table on the workspace, you must specify which one you mean by number or (Q) for query unless you mean the current image. When referring to a display image, query images aren't counted; the number always refers to the nth display image of the table, counting from the first time the image was displayed.

Using a field specifier in an assignment statement works only while Paradox is in Edit or CoEdit mode. In fact, it is equivalent to

1. moving the cursor to the specified field of a table
2. clearing the field
3. typing the new value
4. returning the cursor to its original position

The type of value assigned must be compatible with the type of field to which it is assigned; you can't assign a date to a numeric field, for example. You can mix assignments for number and currency fields freely however. In the case of strings, the field being assigned to must

be large enough to hold the string; otherwise, the string will be truncated to fit the field.

---

### **Moving to specific fields**

You can use a field specifier in a MOVETO command to move the cursor to a given field (see the description in the *PAL Reference* for details). For example,

```
MOVETO[Orders(Q)-> Date] ; moves to Date field in query
                          ; image for Orders
MOVETO FIELD "Date"      ; with FIELD keyword, moves to named field
                          ; in current image
MOVETO [Date]            ; moves to field in current image
```

In addition, you can use MOVETO to move to a certain image or record number:

```
MOVETO "Customer"       ; moves to Customer table on workspace
MOVETO RECORD 55        ; moves to 55th record in current image
```

---

### **Functions**

Functions look like commands—keywords that take arguments—but are really built-in formulas that return a value to your script or expression. PAL provides an extensive library of built-in functions that

- search for, manipulate, and format strings
- manipulate dates and times
- perform trigonometric and logarithmic calculations
- compute statistics, such as variance and standard deviation
- perform financial calculations, such as net present value
- convert values from one data type to another
- perform screen, keyboard, printer, and file input and output
- obtain information about tables, images, windows, the Editor, and the current state of Paradox

You can also define your own functions by using procedures, which are described in Chapter 6. You use functions and procedures, by calling them—typing the name of the function followed by a list of arguments in parentheses. Since functions return values, they are almost always used on the right side of an assignment statement. For example,

```
Num = SQRT(49)           ; Num is set to 7, since that is
                          ; square root of 49
Str = UPPER("abc")      ; Str is set to "ABC", the uppercase
                          ; image of "abc"
Tym = TIME()            ; Tym is set to the current system time
Result = UPPER(GETCHAR()) ; Next character typed at keyboard is converted
                          ; to uppercase and stored in Result
NewDate = Future(TODAY()) ; NewDate assigned value returned
                          ; from procedure named future which in
                          ; turn is passed the current system date
```

**Note** The parentheses following the name of the function must be provided, even if that function takes no arguments. The last two entries in the preceding example show that the arguments can be in the form of any valid expression, including another function call.

The function reference in the *PAL Reference* contains a full list of PAL functions along with a complete description of each. Some are also discussed in the following section, “Manipulating values.”

---

## Manipulating values

---

### String manipulation

You can manipulate Paradox alphanumeric values and memos (strings) directly with the + operator, a process known as concatenation because the two strings are joined (concatenated) to form one. For example,

```
Pgm = "Pair a " + "docks"
```

results in the string “Pair a docks” stored in the variable Pgm. You can concatenate strings up to the 64 million character alphanumeric limit. You can also use the comparison operators with strings. For both the = and <> operators, Paradox checks for exact matches between the strings. For instance, using the string in the previous example, you could test two strings like this:

```
IF (Pgm = "Paradox")
  THEN MESSAGE "By jove, they're the same!"
  ELSE MESSAGE "Sorry, no match."
ENDIF
SLEEP 2000
```

If Pgm contains “Pair a docks”—or even “Paradox ” (note the space at the end)—there is no exact character-by-character match. For two strings to be equal, they must be the same length and have all matching characters.

You can also use the < (less than) and > (greater than) comparison operators with strings. As with all string comparisons, PAL examines the first character of each string and checks to see if they’re the same. If they are, PAL moves on to the next character and performs the test again, and continues until the first non-matching character is found. String ordering is done according to the current PARADOX.SOR file:

```
"A" < "B"           ; for all sort orders
"A" < "a"           ; for ASCII order only
"Pair a docks" < "Paradox" ; at the first non-matching character,
                       ; "i" < "r"
```

You can also manipulate strings with the PAL functions shown in Table 3-8. They are described in detail in the *PAL Reference*.



Table 3-8 String functions

Type	Function	Action
Conversion	ASC()	Converts character to ASCII number
	CHR()	Converts ASCII number to character
	DATEVAL()	Converts string to a date value
	LOWER()	Converts all characters to lowercase
	NUMVAL()	Converts string to a numeric value
	STRVAL()	Converts any data type to a string
	UPPER()	Converts all characters to uppercase
Information	LEN()	Returns length of string
	MATCH()	Tests whether string matches a pattern
	SEARCH()	Returns position of substring in string
	SEARCHFROM()	Starts at a specific offset and returns position of substring in string
Miscellaneous	FILL()	Fills string with specified number of specified characters
	FORMAT()	Formats string in specified way
	SPACES()	Fills entire string with spaces
	SUBSTR()	Extracts substring from string

**Note** The NUMVAL() and DATEVAL() functions convert strings into numbers and dates. If you want to perform an arithmetic operation on a string that contains a number or date, use one of these functions to make the string the appropriate data type and then perform the operation; afterwards, you can use STRVAL() to reconvert the result to a string. For example,

```

TextNum = "209"
TextNum = STRVAL (NUMVAL(TextNum) - 35)

```

Conversion function  
performed first

Arithmetic operation  
performed next

Reconversion  
performed last

## Date manipulation

### Date formats

PAL supports 13 different date formats for output, as shown in Table 3-9.

Table 3-9 Date formats

Format	Date mask	Example output
1	mm/dd/yy	7/22/89
2	Month dd, yyyy	July 22, 1989
3	mm/dd	7/22
4	mm/yy	7/89
5	dd-Mon-yy	22-Jul-89
6	Mon yy	Jul 89
7	dd-Mon-yyyy	22-Jul-1989
8	mm/dd/yyyy	7/22/1989
9*	dd.mm.yy	22.07.89
10*	dd/mm/yy	22/07/89
11*	yy-mm-dd	89-07-22
12*	yy.mm.dd	89.07.22
13*	dd/mm/yyyy	22/07/1989

\* Formats not available in Paradox versions 1.0 or 1.1.

You can use the `FORMAT()` function (see Chapter 5) to display or print dates in any of these formats regardless of how the date values happen to be stored in the table. To print a date in format 7, for instance, you would use the format specifier "d7" in your `FORMAT()` statement:

```
d = FORMAT("d7", 7/22/1987) ; d = 22-Jul-1987
```

or with a field specifier

```
d = FORMAT("d7", [Date Hired])
```

In Table 3-9, notice how spaces are used in the first position, and zeros elsewhere, to fill out single-digit month and day values. This applies to all formats except 2, where no effort is made to align values.

### **Date operations**

As long as dates are expressed in one of Paradox's four interactive date formats (mm/dd/yy, dd-Mon-yy, dd.mm.yy, and yy.mm.dd), you can use them in calculations. Use the + and - operators (see Table 3-4) to compare two dates and find the number of days between them:

```
SendCards = 12/25/92 - 12 ; set SendCards to 12 days before
                          ; Christmas, 1992
PkgReturns = 12/25/92 + 7 ; set PkgReturns to 7 days after
                          ; Christmas, 1992
Shopping = 12/25/92 - 1-Sep-92 ; set Shopping to number of days
                              ; between September 1 and Christmas,
                              ; 1992
```

You can use comparison operators with dates even if they are in different formats. Earlier dates are considered “less than” later dates. The \* and / operators do not apply to dates.

---

## Date functions

You can manipulate date values with the PAL functions shown in Table 3-10. They are described in detail in the *PAL Reference*.

Table 3-10 Date functions

Function	Action
BLANKDATE()	Generates a blank date
DATEVAL()	Converts string to date value
DAY()	Returns day of the month
DOW()	Returns day of the week as a string
MONTH()	Returns numeric value of month
MOY()	Returns month of the year as a string
STRVAL()	Converts date value to a string
TICKS()	Returns the time of day as an integer
TIME()	Returns current time*
TODAY()	Returns current date*
USDATE()	Returns the specified date in U.S. format
YEAR()	Returns four-digit year value

\* Based on computer’s system clock.

The date functions expect one of the four interactive date formats in their arguments.

---

## Date patterns

To ZOOM, LOCATE, or query for dates in a particular month or year, you can use the wildcard operators described in Chapter 5 of the *User’s Guide*. The @ operator stands for any single character, while the .. operator stands for a series of any number of characters. For example, you could enter these conditions in the date field of a query form:

```

../..63      ; to search for any date in 1963
6/..63      ; to search for any date in June of 1963
../..6@     ; to search for any date in the 1960’s

```

The pattern used in the query form must reflect the current default interactive date format, as set using the Custom Configuration Program. Image settings that override the default for an individual column have no effect. For example, if the default date format were dd-Mon-yy, the examples shown previously would not work.

Although you can’t use wildcard operators in PAL date expressions, you can achieve the same effect with date functions, as in the following fragments of PAL programs:

```

IF (YEAR([Date]) = 1963)           ; to search for any date in 1963
...

IF (MONTH([Date]) = 6
   AND (YEAR[Date]) = 1963)       ; to search for any date in
                                   ; June of 1963
...

IF (YEAR([Date]) >= 1960
   AND (YEAR([Date]) <= 1969)     ; to search for any date in
                                   ; the 1960's
...

```

---

## PAL keycodes

Many PAL commands and functions, such as ASC(), KEYPRESS, SETKEY, SHOWMENU, WAIT, and others, take special PAL keycodes as arguments. Using these keycodes, you can represent any valid key or key combination on the keyboard, as well as any character in the regular ASCII and IBM extended characters set.

Keycodes can take any of the following forms:

- single-character strings
- strings representing function key names
- strings representing special key names
- positive numbers representing ASCII codes
- negative numbers representing IBM extended codes

As shown next, many keys can be referred to in several ways, some keys in only one. The most convenient method often depends on the context. All valid PAL keycodes are listed in the *PAL Reference*.

---

### Single-character strings

You can refer to all letters, numerals, and typeable symbols on the IBM keyboard using a single-character string that is the same as the character on the key cap. For example,

```

"A"   ; refers to capital A
"a"   ; refers to lowercase a
"5"   ; refers to numeral 5 (either on main keyboard or numeric keypad)

```

Using quotation marks as shown above, you can refer to the following symbols in addition to the letters and numbers:

```

! @ # $ % ^ & * ( ) _ - + = { } [ ] : ; ' ~ < , > . ? / \ " . ,

```

You can hold down the *Alt* key and use the numeric keypad to enter ASCII characters that can't be typed directly. For example, to enter the extended ASCII code 224, hold down the *Alt* key and type **224** on the numeric keypad. This will appear on your screen as "α".

You can use backslash codes for certain keys:

```

"\t"      ; refers to tab [Ctrl][I]
"\n"      ; refers to newline [Ctrl][J]
"\f"      ; refers to formfeed [Ctrl][L]
"\r"      ; refers to carriage return [Ctrl][M]
"\""      ; refers to double quotation mark "
"\"       ; refers to backslash \

```

To reference any other key, you must use one of the four alternate methods described next.

---

### **Function keys**

You can use the strings "F1" through "F40" to refer to these function keys and combinations:

- "F1" through "F10": function keys *F1* through *F10* pressed alone; there are no string equivalents for *F11* and *F12* because Paradox does not use them.
- "F11" to "F20": function keys pressed with *Shift*.
- "F21" to "F30": function keys pressed with *Ctrl*.
- "F31" to "F40": function keys pressed with *Alt*.

For example,

```

"F1"      ; refers to [F1] (normally Help)
"F13"     ; refers to [Shift][F3]
"F31"     ; refers to [Alt][F1]

```

---

### **Special Paradox key names**

Paradox special keys, such as Do-It! *F2*, Rotate *Ctrl-R*, and *Home* have equivalent script names shown in Table 2-1 in Chapter 2. You can use these names to refer to them in keycode expressions. For example,

```

DO_IT!    ; refers to Do-It! [F2]
INSTANTRECORD ; refers to Instant Script Record [Alt][F3]
ROTATE    ; refers to Rotate [Ctrl][R]
HOME      ; refers to [Home]

```

Case (upper or lower) doesn't matter when typing these special key names.

---

### **Positive numbers representing ASCII codes**

ASCII codes are the standard way of defining the fixed set of characters that can be represented by the IBM PC and PC compatibles. You can use any of the 256 positive decimal number codes in the ASCII character set (and the extended character set) to refer to the corresponding character. For example,

```

65        ; refers to uppercase A
18        ; refers to [Ctrl][R] (ROTATE in Paradox)
249       ; refers to [Alt]249 (•)

```

The complete set of ASCII codes and the characters they represent is listed in Appendix H of the *PAL Reference*.

---

**Negative numbers  
representing IBM extended  
codes**

IBM has established an additional code set to refer to significant keys and keystroke combinations not represented by the 256 codes in the ASCII character set. These IBM extended codes refer to:

- all direction keys
- *Alt*-key combinations
- function keys *F1* through *F10*, pressed alone or combined with *Shift*, *Ctrl*, or *Alt*

The extended key codes are represented by a positive decimal number between 1 and 132. To refer to one of these keys in PAL, simply make the number negative. For example,

```
-30           ; refers to [Alt][A] (IBM extended code 30)
-71           ; refers to [Home] (IBM extended code 71)
-104          ; refers to [Alt][F1] (IBM extended code 104)
```

Be careful to specify a negative number. Positive numbers refer to regular ASCII codes, negative numbers to IBM extended codes. The complete keycodes for the ASCII and IBM extended character sets available in PAL programs is listed in Appendix G of the *PAL Reference*.

---

**Using keycodes in  
expressions**

There are several ways to refer to most keys. For instance, all of the following examples let users examine a table on the workspace until they press Do-It! *F2*:

```
WAIT TABLE UNTIL "F2"           ; expressed as a quoted function key
WAIT TABLE UNTIL "DO_IT!"       ; expressed as the name of a special key
WAIT TABLE UNTIL -60            ; expressed as an IBM extended code
```

Alternate forms of the same keycode are listed in Appendix G of the *PAL Reference*.

You can also represent a key with an expression that returns any valid keycode. For example,

```
x = "2"
WAIT TABLE UNTIL "F" + x        ; expression results in function
                                ; key name "F2"
```

---

## Converting codes

The GETCHAR() function can be used to trap the next key the user presses. GETCHAR() returns the keycode as a positive or negative ASCII code, however, which is not always convenient or readable in a comparison statement. You can use the ASC() function to convert other keycode formats into ASCII numbers. For example,

```
MenChar = ASC("Menu")           ; get PAL keycode for Menu [F10]
IF (GETCHAR() = MenChar)        ; compare it against next character typed
    THEN MESSAGE "User pressed Menu."
    ELSE MESSAGE "User didn't press Menu."
ENDIF
SLEEP 2000
MESSAGE ""
```

---

### Example 3-3 Branching based on the next keystroke

---

You can use ASC() and GETCHAR() with the SWITCH and CASE commands to branch to different portions of a script depending on the next key a user types.

As you'll see in Chapter 14 and Chapter 15, it is far easier to use a menu or a dialog box to interact with a user. We've provided this example to show you how you can alternate between methods of referring to a keystroke within a script. Here, the first CASE statement uses a special key name, the second a function key name, the third a single-character string, and the last a standard ASCII code to define the key to look for.

Although normally you would not mix types like this within your scripts, this example shows the range of possibilities.

```
c = GETCHAR()           ; wait for a character
SWITCH
CASE c = ASC("Menu"):   ; Menu key pressed?
    PLAY "Menusels"    ; yes, so play script Menusels

CASE c = ASC("F11"):    ; Shift-F1 keys pressed?
    PLAY "Showhelp"    ; yes, so play script Showhelp

CASE c = ASC("Q"):      ; Q (for Quit) key pressed?
    QUIT               ; yes, so quit

CASE c = 113:          ; 113 = ASCII q (lowercase Q)
    QUIT               ; yes, so quit
ENDSWITCH
```





# Control structures

When you use Paradox interactively, you determine the order in which things are done. You select what to do next from the various Paradox menus and PAL commands available to you. Depending on what happens, you can decide to change and rearrange tasks as needed.

PAL normally executes commands in scripts in their order of appearance in the script. But you can use PAL's *control structures* to change and rearrange the flow of control in a script—the order in which commands are executed. These control commands give your applications the ability to make the same decisions you would make if you were using Paradox interactively.

PAL's control structures include

- *branching commands*, which perform specific commands depending on whether certain conditions are met
- *looping commands*, which repeat a series of commands until a certain condition is met
- *termination commands*, which return from procedures and leave scripts

This chapter gives an overview of PAL's control structures and when to use each. The complete syntax of each command is described in the command reference in the *PAL Reference*.

---

## Conditions

To determine which commands to execute next, several of PAL's control commands involve testing a condition, which simply results in a logical value or *expression*. For example, the following expressions could be used as conditions:

```

x = 4 ; tests whether variable x is 4
Age > 21 ; tests whether variable Age is greater than 21
[Date Hired] < TODAY() - 90 ; tests whether employee was hired more
; than 90 days ago

```

The condition portion of a control command tells PAL whether to execute a series of commands, as shown in the syntax of the IF command:

```

IF Condition
  THEN Commands ; execute commands only if Condition is True
ENDIF

```

or which series of commands to execute, as shown in the syntax of the SWITCH command:

```

SWITCH
  CASE Condition1 : Commands1 ; execute Commands1 for Condition1
  CASE Condition2 : Commands2 ; execute Commands2 for Condition2
  ...
ENDSWITCH

```

Other control commands perform their actions immediately and require no conditions to guide them in what to do next, as in RETURN and QUITLOOP.

## Branching

*Branching* lets a PAL script choose one set of commands to execute from among several you specify. The choice is based on a condition that exists when that portion of the script is played. Paradox menus are good examples of branching, since your choice determines which portion of the Paradox program is invoked next. If you choose Modify | Sort, for example, Paradox begins executing the commands that tell it how to sort tables.

Branching within PAL works the same way: you specify a condition to test and the commands to execute if the condition is true. There are two branching commands you can use:

- IF, when there are only two ways to branch
- SWITCH, when there are multiple ways to branch

To branch and execute a procedure, you can use

- dynamic arrays and EXECPROC

There is also a branching function

- IIF(), for branching within a statement

In practice, the distinction between IF and SWITCH is more complicated than two branches versus many, since you can nest one structure within another. For instance, one of the commands that can

be executed within an IF structure is another IF structure, effectively resulting in a three-way branch. PAL supports unlimited nesting of control structures. SWITCH is more appropriate for true multiple branches, while nested IFs are preferred for hierarchical two-way branches. For large menu trees that execute procedures, using EXECPROC with a dynamic array provides a clear performance advantage over SWITCH.

If you have programmed in BASIC or certain other languages, you may notice the absence of a GOTO command. You can achieve the same results with PAL's alternatives, but in a much more organized and structured way.

---

## IF

An IF statement is simply an either/or switch. If some condition is true, then Paradox performs certain statements. If the condition is false, Paradox might perform other statements. IF structures look like this:

```
If Condition
  THEN Commands1
  ELSE Commands2      ; an ELSE clause is optional
ENDIF
```

The ELSE clause in the IF command is optional. If you omit it and the condition is false, PAL moves directly to the command following ENDIF. The ENDIF clause *must* appear at the end of the IF structure.

### Example 4-1 IF...THEN...ELSE branching

---

This example shows the two-way branching capability of IF. If the specified table exists, the example plays the GoCust script. If the table does not exist, the example plays the GetTab script. In either case, when the appropriate script finishes, the example resumes with whatever follows the ENDIF keyword.

```
IF ISTABLE("Customer")
  THEN PLAY "GoCust"
  ELSE PLAY "GetTab"
ENDIF
```

You can nest IF...THEN...ELSE statements like this:

```
IF Condition1
  THEN
    IF Condition2      ; beginning of nested IF
      THEN Commands1
      ELSE Commands2
    ENDIF              ; end of nested IF
  ELSE Commands3
ENDIF
```

Although this example shows an IF...THEN...ELSE statement nested within a THEN clause, you could also nest one within an ELSE clause. Each ENDIF in a nested IF construct ends the most recently active IF. In this example, the first ENDIF ends the second IF, while the second ENDIF ends the first (the only active IF left).

In cases of true multiple branching, SWITCH is usually more effective than repeated nested IFs. However, if the multiple branches are actually hierarchical two-way branches, nested IFs are preferred. For example,

```
IF A
  THEN
    IF B
      THEN Commands1
      ELSE Commands2
    ENDIF
  ELSE
    IF B
      THEN Commands3
      ELSE Commands4
    ENDIF
  ENDIF
```

is preferable to

```
SWITCH
  CASE (A AND B)      : Commands1
  CASE (A AND NOT B) : Commands2
  CASE (NOT A AND B)  : Commands3
  CASE (NOT A AND NOT B) : Commands4
ENDSWITCH
```

In the example using nested IFs, Paradox needs to test only two conditions. In contrast, in the example using SWITCH, Paradox might have to test up to eight conditions, depending on which of the CASE statements is true. To optimize a SWITCH statement, organize the CASE statements so that the condition most likely to be true is first, the second most likely is second, and so on.

---

## SWITCH

The SWITCH command is like multiple IF...THEN statements. Use it to branch to one of multiple places in your script when the simple either/or branching of IF...THEN is not appropriate. You can use SWITCH with menus or in conjunction with the SHOWMENU command to produce the effect of Paradox-style menus in an application.

A SWITCH includes one CASE statement for each possible branch. The CASE statement contains a condition, followed by a colon and a set of commands to be executed if the condition is true. You can also include an OTHERWISE branch to be executed if none of the CASE conditions is true. Only one set of commands is executed per pass through the SWITCH. The syntax is

```
SWITCH
  CASE Condition1 : Commands1
  CASE Condition2 : Commands2
  .
  .
  .
  CASE Condition-n: Commands-n
  OTHERWISE: Commands
ENDSWITCH
```

## Example 4-2 SWITCH branching from a menu

---

Suppose you want to present a Paradox-style menu with six choices—add data, delete data, change data, save data, retrieve data, and exit. You can use the SHOWMENU command to display the menu and the SWITCH command to branch to the appropriate section of code.

In this example, the OTHERWISE clause covers the possibility that the user presses *Esc*, which is a valid response to a Paradox-style menu, without actually making a selection. Notice how the use of indentation makes it easy to see which commands are executed for each CASE.

```
; Display the menu

SHOWMENU
  "Add"      : "Add data",
  "Delete"   : "Delete data",
  "Change"   : "Change data",
  "Save"     : "Save data",
  "Load"     : "Load data",
  "Exit"     : "Leave the program"
TO response                                     ; store choice in response

; Now, based on response, execute appropriate script

SWITCH
CASE response = "Add":
  PLAY "Addat"
CASE response = "Delete":
  PLAY "De1dat"
CASE response = "Change":
  PLAY "Chgdat"
CASE response = "Save":
  PLAY "Savdat"
CASE response = "Load":
  PLAY "Loaddat"
CASE response = "Exit":
  QUIT
OTHERWISE:                                     ; in case of Esc
  BEEP
  MESSAGE "If you want to exit, choose Exit"
ENDSWITCH
```

Since a menu is a true multiple branch, you wouldn't want to use nested IFs in this case. If you did, you'd end up with:

```
IF response = "Add"
  THEN PLAY "Addat"
ELSE
  IF Response = "Delete"
    THEN PLAY "De1dat"
  ELSE
    IF ...
```

and so on. Not only is this structure more confusing, but it also increases your chance of mismatching IFs and ENDIFs.

You can use nested SWITCH commands to build complex branching structures. Or you can use SWITCH to control what happens when values fall into specific classes, such as:

```

SWITCH
CASE Number < 0 :
    Number = ABS(Number)
CASE Number > 0 :
    MESSAGE "Thank you for the number"
    SLEEP 2000
CASE Number = 0 :
    MESSAGE "Can't operate with value of zero"
    SLEEP 2000
ENDSWITCH

```

Here SWITCH was used to test a value stored in *Number* to see if it was negative or zero.

---

## Branching with dynamic arrays

When you want to branch and execute a different procedure based on multiple conditions, dynamic arrays provide a substantial performance advantage over a SWITCH with multiple CASE statements. The CASE statements within a SWITCH are processed linearly and evaluated one-at-a-time; when there are many CASE statements (and the one you need is near the end!), your processing can be delayed. Because dynamic arrays use associative memory, the lookup process is almost instantaneous.

Like SWITCH branching, dynamic array branching is frequently used in conjunction with SHOWMENU, SHOWPOPUP, and SHOWPULLDOWN. Each tag in the dynamic array represents a possible value of the variable in the menu command; each element evaluates to the name of a procedure. You can also include an IF branch to be executed if none of the conditions is true.

### Example 4-3 Dynamic array branching

---

You can rewrite the command in Example 4-2 to use dynamic array branching instead of a SWITCH. In this example, the IF command covers the possibility that the user presses *ESC* without actually making a selection.

```

; define the dynamic array

DYNARRAY MenuTask []
MenuTask ["Add"] = "Addat"
MenuTask ["Delete"] = "Del1dat"
MenuTask ["Change"] = "Chgdat"
MenuTask ["Save"] = "Savedat"
MenuTask ["Load"] = "Loaddat"
MenuTask ["Exit"] = "QuitDat"

; now display the menu

SHOWMENU
    "Add" : "Add data",
    "Delete" : "Delete data",
    "Change" : "Change data",
    "Save" : "Save data",
    "Load" : "Load data",
    "Exit" : "Leave the application"
TO Response

; now execute appropriate procedure based on response

```

```

IF ISASSIGNED(MenuTask[Response])
  THEN EXECPROC MenuTask[Response]
ELSE BEEP
  MESSAGE "Select Exit if you want to leave."
  SLEEP 1500
  MESSAGE ""
ENDIF

```

---

## IIF ( )

The immediate if function IIF() allows for branching within a single statement:

```
IIF (Condition, ValueIfTrue, ValueIfFalse)
```

You can use IIF() anywhere you would supply an expression; for instance, in calculated fields on forms or reports.

Many IF structures can be rewritten as IIF() statements, for example

```

IF ISBLANK([Qty])
  THEN MESSAGE "0.00"
  ELSE MESSAGE [Qty] * [Price]
ENDIF

```

can be rewritten

```
MESSAGE IIF(ISBLANK([Qty]), "0.00", [Qty] * [Price])
```

The IF structure in Example 4-1 could be rewritten as:

```
PLAY IIF(ISTABLE("Customer"), "GoCust", "GetTab")
```

---

## Loops

A *loop* is a set of commands that are executed repeatedly until some condition is met. PAL supports four loop structures:

- FOR repeats the loop until a counter variable has hit the maximum or minimum value specified.
- FOREACH is a special kind of FOR loop used with dynamic arrays.
- WHILE repeats the loop as long as a condition specified is True.
- SCAN repeats the loop for specified records of a Paradox table.

In addition, the LOOP and QUITLOOP commands let you interrupt or exit from loops.

Like branching commands, loops can be nested within one another (you could have a WHILE loop inside of a FOR loop, for example). You can also use loops in conjunction with branching commands to create more complex patterns, such as "execute this loop *x* number of times if the condition is met and *y* number of times if not":

```

WINDOW CREATE TO DisplayWindow      ; create window to display output

IF Condition = True
  THEN
    FOR Count FROM 1 TO 10
      ?? Count                       ; True prints 1 2 3 4 5 6 7 8 9 10
    ENDFOR
  ELSE
    FOR Count FROM 1 TO 5
      ?? Count                       ; False prints 1 2 3 4 5
    ENDFOR
ENDIF

```

In the following case, the IIF() function produces the same effect with far fewer lines of code:

```

WINDOW CREATE TO DisplayWindow

Limit = IIF(Condition, 10, 5)

FOR Count FROM 1 TO Limit
  ?? Count
ENDFOR

```

---

## FOR

A FOR loop uses a numeric variable to control the number of times the loop is executed. You set the variable to a beginning value and the commands in the loop are executed. Then the variable is incremented or decremented by a step you specify and the commands are repeated; this continues until the variable exceeds a maximum or minimum value you specify. The syntax is

```

FOR Counter
  [ FROM Expression1 ]
  [ TO Expression2 ]
  [ STEP Expression3 ]
  Commands
ENDFOR

```

*Counter* is a numeric or date variable, and ENDFOR must be used to end the structure. You can use FOR without the FROM, TO, and STEP keywords:

- If the FROM value is omitted, the counter starts at the current value of *Counter*.
- If the TO value is omitted, the FOR loop executes indefinitely.
- If STEP is omitted, the counter increments by 1 each time through the loop.

FOR is especially useful when you

- know how many times you want to execute the loop. For example, this FOR loop blanks out the top five lines on the canvas:

```

FOR j FROM 0 TO 4
  @ j,0

```



```
CLEAR EOL
ENDFOR
```

- want to use the incremented value in the loop. This example uses the loop counter to print the squares of the first ten digits:

```
FOR Counter FROM 1 TO 10
  ? (Counter * Counter) ; print number squared
ENDFOR
```

- want to use variables to define the starting, ending, and step values. This example uses variables to square the same numbers:

```
x = 1
y = 10
FOR Counter FROM x TO y STEP x
  ? (Counter * Counter)
ENDFOR
```

You can also use a FOR statement to go from a beginning value down to a minimum value. To do so, specify a negative value for STEP. For example, this FOR loop performs a countdown on the canvas:

```
; Countdown

ECHO NORMAL
WINDOW CREATE TO DisplayWin ; create canvas to display output
FOR Counter FROM 9 TO 0 STEP -1 ; set the step to -1
  @ Counter, 3 ?? "T - ",Counter ; use counter to count down
  SLEEP 1000
ENDFOR
?? " ... Blastoff!"
```

The FOR statement lets you step through the elements in a fixed array. The following example shows how to print each element in a fixed array:

```
i = ARRAYSIZE (FArray)
FOR x FROM 1 to i
  ? FArray[x]
ENDFOR
```

Use the FOREACH statement to iterate through the elements of a dynamic array.

---

## FOREACH

PAL provides a special control structure, FOREACH...ENDFOREACH, to step through the elements of a dynamic array. You cannot use the FOR command to step through a dynamic array because the tag names of the elements are not necessarily sequential integers.

To iterate over a dynamic array, you can use the control structure FOREACH, as shown in the following example:

```
FOREACH x IN FArray
  ? x, " ", FArray[x]
ENDFOREACH
```

Because the elements of a dynamic array are unordered, the FOREACH command will step through them in an unpredictable order. If you add elements to a dynamic array within a FOREACH loop, however, the current FOREACH command will not iterate over those new elements.

---

## WHILE

Use a WHILE loop to repeat a group of commands an indeterminate number of times. The loop continues as long as a certain condition returns True. The syntax is

```
WHILE Condition
    Commands
ENDWHILE
```

If the condition becomes false, PAL stops executing the commands within the loop and skips to the first command beyond the ENDWHILE statement.

The condition test occurs only when the commands in the loop have been executed and the loop is about to be repeated. That is, the condition test is not continuous, but is performed once at the beginning of the loop and again just prior to the ENDWHILE, each time control passes through the loop.

A few simple examples illustrate how WHILE loops work:

```
WHILE True           ; executes forever because condition never
    ? "Hello there." ; becomes false
ENDWHILE
```

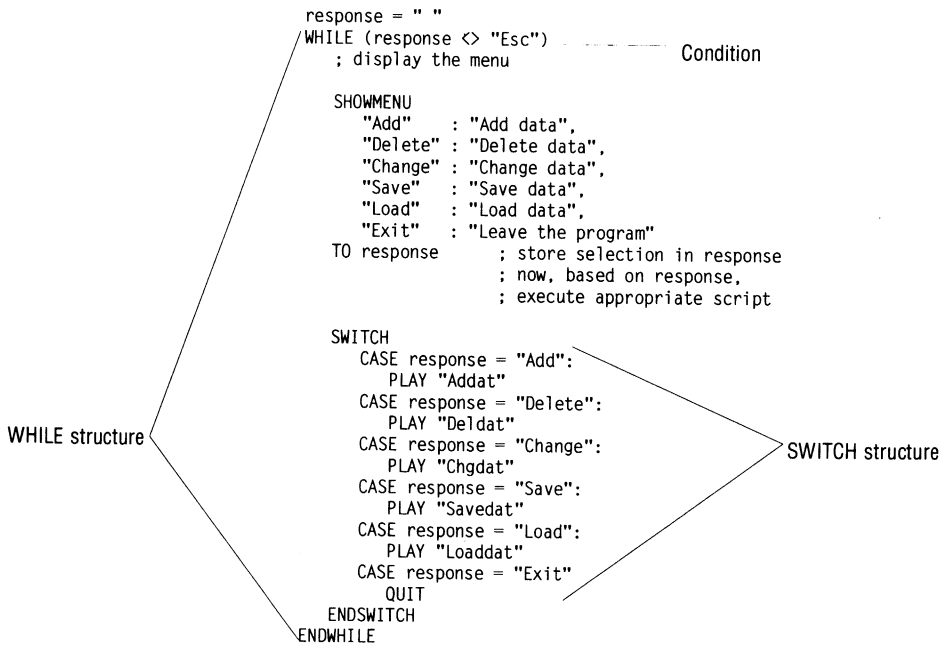
```
WHILE False         ; never executes because condition is never true
    ? "Hello there."
ENDWHILE
```

```
x = True
WHILE x             ; executes once
    ? "Hello there."
    x = False       ; terminate loop condition
ENDWHILE
```

---

### Example 4-4 Using WHILE to redisplay a menu

The SHOWMENU and SWITCH commands in Example 4-2 displayed a menu and played another script depending on the choice. When the other script was finished, PAL resumed with the command following the ENDSWITCH keyword. But if you want the menu to be redisplayed, you could embed the SHOWMENU and SWITCH commands in a WHILE loop. (This is the preferred way to return control to a menu after the commands under one of its selections have been executed.)



As in Example 4-2, when the second script is finished, PAL skips to the end of the SWITCH structure—but it's still in the WHILE loop, so the loop is repeated. Only two things would prevent the repeat: choosing Exit (which ends the script) or pressing *Esc* (since the condition would no longer be true). Notice that the OTHERWISE clause (which previously handled *Esc*) has been omitted. Now if the user presses *Esc*, PAL exits from the loop and continues after the ENDWHILE keyword.

## SCAN

The SCAN command is a control structure that acts on an entire table. It moves through the current table on the workspace, executing a group of commands each time it finds a record that meets a certain condition.

The syntax for SCAN is

```

SCAN
  FOR Condition
  :
  Commands
ENDSCAN
; FOR clause is optional
; : separator is optional
; do these commands for every record in the table
; or for every record that meets FOR condition

```

SCAN is useful when you want to perform the same action on all or a selected set of records in the table. SCAN is a powerful prototyping tool. You can write a command sequence to modify a single record, then put the sequence in a SCAN loop to perform the same action on every record in a table.

A SCAN command without a FOR clause performs actions on every record in the table.

Because FOR is an optional keyword to the SCAN command, the : separator is used to start a FOR loop immediately following a SCAN. For example,

```
SCAN : ; use the : after the SCAN keyword
FOR i FROM 1 TO 10 ; and the FOR clause is treated as a
ENDFOR ; separate command
ENDSCAN
```

#### Example 4-5 Scanning a table

---

Suppose you discover that, while adding names to your company's mailing list, an operator who didn't know the state code for Nevada had typed "Nevada" in the State field—147 times. You can use a SCAN loop to fix it.

You could also use a CHANGETO query to achieve the same effect.

```
COEDIT "Maillist"
SCAN FOR [State]="Nevada"
      [State] = "NV"
ENDSCAN
DO _IT!
```

---

## Interrupting a loop

Use LOOP and QUITLOOP to repeat or terminate a FOR, FOREACH, WHILE, or SCAN loop.

- LOOP tells PAL to skip the remaining commands in a loop and go back to the beginning of the loop. In a FOR loop, LOOP increments the counter; in a FOREACH loop, it advances to the next element in the dynamic array; in a WHILE loop, it checks the condition again; in a SCAN loop, it moves to the next record.
- QUITLOOP tells PAL to immediately terminate the loop regardless of the loop condition.

### Example 4-6 Interrupting a loop

---

Here's a loop that would never end if it weren't for the QUITLOOP command.

This example—which does *not* show good programming practice—prints the squares of even numbers from 2 to 10.

Notice that the FOR loop doesn't execute the prescribed 10 times because QUITLOOP terminates it after the fifth cycle. Also, the premature loop forced by the LOOP statement means that the MESSAGE never gets displayed.

```
FOR Num
  FROM 2 TO 20 STEP 2
  ?? (Num * Num)           ; displays Num squared
  IF (Num = 10)
    THEN QUITLOOP         ; if 10, stop
    ELSE LOOP              ; if not 10, go back to beginning
  ENDF
  MESSAGE "This line never gets displayed!"
ENDFOR
```

---

## Returning and exiting

*Returning* means going back from the current script or procedure to the higher-level script or procedure that called it. *Exiting* means abandoning all script play entirely. The three commands that perform these functions are

- RETURN, which returns from a script or procedure
- QUIT, which stops script play and returns to Paradox
- EXIT, which stops script play and returns to DOS

---

### RETURN

Use RETURN to finish a procedure or script before the last command and return to the script that called it. PAL ignores any further commands in the procedure or script.

If appropriate, you can also use RETURN to pass a value to the calling script:

- If a procedure ends with a RETURN, the procedure call itself takes on the returned value.
- If a top-level script ends with a RETURN, the returned value is displayed in the message window when control returns to Paradox.

As a side effect of RETURN, the system variable *RetVal* is assigned the returned value in the calling script or procedure.

Suppose you want a procedure to return the value True if its string argument is a capital letter (or starts with a capital letter), and False if it is not:

```
PROC IsCapLetter(Str)
  IF (Str >= "A" AND Str <= "Z")
    THEN RETURN True ; True is returned to calling script
    ELSE RETURN False ; False is returned to calling script
  ENDIF
ENDPROC
```

RETURN is most useful when you take advantage of PAL's ability to modularize an application into multiple scripts and procedures.

---

## QUIT and EXIT

Both QUIT and EXIT *stop* all levels of script play. The QUIT command ends the script and returns to Paradox, while EXIT ends the script, leaves Paradox, and returns to DOS. QUIT takes an optional argument that is displayed in the message window when you return to Paradox. The use of QUIT is illustrated in Example 4-2 and Example 4-4.

# Formatting data

You can use PAL to format values for input and output by

- ❑ using *pictures* to control values that users enter
- ❑ using *format specifications* to control values that are output by scripts

This chapter describes these two formatting methods.

---

## Using pictures to format input

A PAL picture is a powerful and flexible tool for controlling what a user can type into a field during data entry. You can use pictures in ACCEPT and WAIT commands to

- ❑ shape and limit what the user can type into a field
- ❑ make data entry easier by filling in required or default values automatically

You can think of pictures as a way to define new field types by imposing restrictions on existing ones. In effect, the Social Security number picture in Example 5-1 defines a new type of alphanumeric field. So would a picture of a telephone number (with or without the area code), or a part number in which X is always the second character. Pictures can also help users fill in default or repetitive values during data entry. For instance, through a picture, you can specify that the X in the part number will be filled in automatically.

### Example 5-1 Using a picture to format input

Suppose you want a user to enter Social Security numbers. Use a picture in an ACCEPT command to make sure the input has the proper format.

```
ACCEPT "A11"
  PICTURE "###-##-####"
  TO SocSec
```

The picture fills in the hyphens automatically and ensures that the user types the proper number of digits—no other characters will be accepted.

Although Example 5-1 shows a picture in a PAL ACCEPT command, you don't have to use PAL to take advantage of pictures. You can use the ValCheck command on Paradox's Edit or DataEntry menu (see Chapter 11 of the *User's Guide*) to set up a picture for any field in a table. Once a picture is established, Paradox enforces it whenever data in the field is entered or edited.

You can use pictures with any field type except memos. For example, you can require that numbers have a certain number of digits or that the user can enter dates only in a certain month.

When a picture is specified, the user must fill it exactly and completely. In Example 5-1, if a user starts by typing a letter, a beep results. If someone tries to leave the field before typing all the digits, Paradox displays the message **Incomplete field** and leaves the cursor in the field. The only way to avoid filling a picture is to leave the field blank—and you can close that loophole by specifying REQUIRED in your ACCEPT command or validity check.

If Auto|Picture is on for a field with a specified picture, the cursor automatically moves to the next field whenever the picture has been filled with a valid entry. This allows you to move your user through a data entry screen with speed and ease.

While a user is typing data into a picture, pressing *Backspace* or *Del* erases characters and *Ctrl-Backspace* clears the field—as long as the resulting entry fills the picture. You can also design the picture to fill in optional characters when the user presses *Space*.

---

## How to define pictures

A picture is a kind of pattern. It consists of a sequence of literal characters interlaced with any of the match characters listed in Table 5-1.

Table 5-1 Match characters in pictures

---

Picture element	Description
<b>Match characters</b>	
#	Accept only a digit
?	Accept only a letter (upper- or lowercase)
&	Accept only a letter, convert to uppercase
@	Accept any character
!	Accept any character, convert to uppercase
†	Any character taken literally



Picture element	Description
-----------------	-------------

**Special characters**

;	Take next character literally
*	Repetition count
[ ]	Option
{ }	Grouping operators
,	Set of alternatives

Any number, letter, or punctuation character not defined as one of the unique match or special characters (that is, anything you can type that isn't on this list) is taken literally.

The Social Security number in Example 5-1 was specified with the picture `### - ## - ####`. The # is a match character that accepts only digits in its place. The hyphen is a literal character, meaning that it must literally appear in the typed value (matched exactly).

Literal characters in a picture are filled in automatically unless

- you specify otherwise (see “Inhibiting automatic fill-in” later in this chapter)
- they occur at the beginning of a picture (this helps to accommodate blank fields)

For example, when the cursor arrives at a blank field governed by the picture

`ABC-###`

the “ABC-” is not filled in automatically because it occurs at the beginning of the picture. But as soon as the user types **A** or **a**, or presses *Space*, the “ABC-” appears.

If you want to specify a literal character that happens to be a match character, precede it with a semicolon (;). For example, here’s how you’d specify a part number that contains three letters, a hyphen, and a number sign (#):

`??-;#`

If you omitted the semicolon, the picture would call for three letters, a hyphen, and a digit. You can use the semicolon in a picture to precede any of the characters in Table 5-1 that you want to be taken literally—including the semicolon itself. Here are some other examples:

<code>\$*#.##;@</code>	; price each, like \$345.67@
<code>*?;?</code>	; questions, adding ? to any text
<code>###;.,###</code>	; six-digit number with comma separating thousands, ; like 345,678

???:;# ; part number with three letters, semicolon, and  
 ; number sign, like ABC;#

## Special features of pictures

The special features of pictures are described in detail in the following sections. These special features are summarized in Table 5-2.

Table 5-2 Special features of pictures

Operator	Name	Examples	Satisfied by
*	Repeat	*5# *# (*#.##), *#.## &*?	Five numbers; equivalent to ##### Zero or more numbers Currency amounts, negative or positive Initial capitalization of a single word
{ }	Set	*5{##} {20,40,60,75,100}W	Ten numbers or five times the set of two numbers Light bulbs in different wattages
[ ]	Optional	###-####[###[#]] *#[*?][@[ ]]	Phone number with or without a 3- or 4-digit extension Any number of capitalized words and initials
,	Alternative	RED, BLUE ##,##	Literals "RED" or "BLUE"; entering an R fills RED, a B fills BLUE 2- or 3-digit number; ##,### would not work
{ }	Inhibit Fill	###{-}##	The hyphen is not filled in automatically; the user must press <i>Space</i> to have the hyphen filled in

## Repetition counts

You'll find the repetition count match character (\*) useful when specifying pictures for long fields. For example, these two pictures are equivalent:

```
*25#
#####
```

The match character \* in the first picture is followed by a number, which is the number of times to repeat the match character or literal that follows. If you omit the \*, the picture would mean the number 25 followed by any other digit.

To repeat a group of match characters, enclose the group in braces { } following the number of repetitions:

```
*3{##:} ; equivalent to ##:##:##:
```

Notice how this picture mixes literal and match characters in the group. Each group consists of two digits to be typed by the user and a colon to be filled in by Paradox.

Also use the grouping braces { } when you want to repeat a single number, as in

`*3{5}#` ; equivalent to `555#` (three fives followed by any other digit)

Without the braces, Paradox would think you wanted `*35#` — 35 repetitions of “#”.

Omitting the number after the \*, tells Paradox to accept zero or more repetitions of the character or group. You might use this variation when you want the user to type at least a certain number of characters, but perhaps more. For example, this picture in a State field

`&&*&`

requires at least the two-letter state abbreviation, but would also accept a state name spelled out in full.

To use the \* as a literal character in a picture, precede it with a semicolon (;). For example,

`;*###;* ; equivalent to *###*`

---

### Optional elements

You can make part of a value optional by enclosing the corresponding part of the picture within brackets [ ]. This means that the user can enter data for that part of the value, but is not required to do so. The user can accept the optional part by typing a character that fits it or by pressing *Space* if the character is a literal.

**Note** When the user reaches an optional part of a value, any matching character accepts the entire optional part. For example,

`[###] ###-####`

*doesn't* make the area code optional. Suppose the user tries to skip the area code and types the first digit of the phone number. Since this matches the first optional character, Paradox assumes it is the area code and requires all 10 digits.

Similarly, the following picture won't work to specify a number with either two or three digits:

`[#]##`

since the first digit typed is always understood to be the [#]. To specify an optional number of digits, place the optional portion *after* the required portion:

`##[#]`

Here, the first two digits typed will match the mandatory # characters, and the third digit may or may not be added. Remember that pictures are always scanned left to right, so the optional portion of a repeating element should always come at the end.

Example 5-2 demonstrates the correct way to create a picture for a telephone number with an optional area code.

### Example 5-2 Options in pictures

---

The following picture accepts a 7-digit telephone number with an optional, parenthesized 3-digit area code:

```
[(###)] ### - #####
```

Optional part of picture

The user can fill in this field in two ways:

- If the first character typed is the left parenthesis or *Space*, it matches the optional part of the picture and Paradox will look for a 3-digit area code before the 7-digit number.
- If the first character typed is a digit, Paradox ignores the option and skips to the required digits. The right parenthesis (if used) and hyphen are filled in automatically.

Literal characters at the beginning of an optional part of a value are never filled in automatically. For example, in the picture

```
#[ABC]#
```

the characters "ABC" are not automatically filled in after the user types the first digit. They *are* filled in if the user types "A" or "a" or presses *Space*. The automatic fill-in occurs only when the filled-in characters are mandatory, or when the user explicitly elects the option by typing the first character or pressing *Space*.

Optional elements can be nested, as in the following picture:

```
#[[#][a]]
```

Because the characters [ and ] are used to specify options, they must be preceded by a semicolon (;) if you want to use them literally in a picture.

---

### Alternative choices

Many applications require the user to type one of several possible choices as part or all of a data value. A part number, for example, might consist of a three-digit number, followed by a color code of RED, GRE, YEL, or BLU, followed by another three-digit number. You specify a set of alternatives in a picture as a comma-separated list of alternatives. For example, you could specify the color-coded part numbers like this:

```
###RED###,###GRE###,###YEL###,###BLU###
```

or, more succinctly,

```
###(RED,GRE,YEL,BLU)###
```

Both of these pictures have the same effect. In the second, the braces have been used to group the alternatives.

As with bracketed options, literal characters at the beginning of an alternative are not filled in automatically. For example, if the user types **345** in the preceding example, the characters **RED** are not then filled in. But if the user then presses *Spacebar* or types *r*, the **ED** is filled in. Similarly, if the user types **g**, the **RE** is filled in, and so on. Paradox always fills in the first matching alternative it finds; that's why **RED** is filled in when the user presses *Space*. To get one of the other options, the user must type its first letter.

### Example 5-3 Alternatives in a date picture

---

The following example lets you restrict input in a date field to dates falling within the first week of a month:

```
#[#]/(1,2,3,4,5,6,7)/###[#][#]
```

Note the use of optional digits to permit months to have either one or two digits and years to have two, three, or four. Again, since pictures are scanned from left to right, the optional elements always come at the end of that part of the picture.

The different alternatives need not be composed of literal characters; they can be picture specifications of any kind (Example 5-3). For example, here's another way of specifying a two- or three-digit number:

```
###.##
```

Again, the picture **##,###** wouldn't do the job, because once the first two digits are typed, the first alternative will be selected and the picture will be fully satisfied. If the user then types a third digit, a beep sounds.

The following picture specifies that an alphanumeric value can either be True or False:

```
True,False
```

Using this picture or variations on it, you can easily obtain the effect of a logical value or create a field of type logical in a Paradox table.

As with other match characters, if you want to use a comma (,) as a literal character in a picture, you must precede it with a semicolon (;).

---

## Inhibiting automatic fill-in

Some users find automatic fill-in of literal characters annoying—particularly if they seldom look at the screen as they type. To inhibit it, simply enclose the literal characters you don't want filled in within braces { }. For example, this picture inhibits the fill-in of hyphens in Social Security numbers:

```
###(-)##(-)###
```

When the user types the first three digits, the hyphen (-) is not filled in. To enter it, the user can press - or *Space*.

Sometimes you must inhibit fill-in to get the effect you want. In the color-coded part numbers in the previous section, suppose we had the color code BRO in addition to RED, GRE, YEL, and BLU. Adding BRO to the preceding picture gives

```
###(RED,GRE,YEL,BLU,BRO)###
```

Now suppose the user types three digits and then a **b**. Since BLU is the first possible match (left to right), Paradox would fill in the LU automatically. This could be wrong, since the user may have had BRO in mind rather than BLU. To solve this problem, use either of these pictures to inhibit automatic fill-in of the L:

```
###(RED,GRE,YEL,B{L}U,BRO)###
```

```
###(RED,GRE,YEL,B{LU,RO})###
```

The second picture, in effect, factors the letter *B* out of both BLU and BRO. This forces the user to type two characters to specify the blue or brown color code. The {LU, RO} is a set of alternative choices that has been nested within the larger set of choices.

Going a step further, here's how you could inhibit all of the fill-in:

```
###(R{E}{D},G{R}{E},Y{E}{L},B{L}{U},B{R}{O})###
```

---

## Picture examples

By using PAL's regular and special match characters, you can create an almost endless variety of pictures—in effect, new data types—for your applications. Here are just a few examples:

```
(*#.##),*#.##           ; currency amounts, using parentheses for  
                          ; negative quantities  
#[#][#]*{;.###}        ; positive integers with commas separating  
                          ; groups of three digits  
{20,40,60,75,100}W      ; light bulbs in different wattages  
{A,B,C,D,E,F,G,H}[R]##-## ; tire sizes  
&*?                     ; initial capitalization of a single word  
&. &. &*?                ; two initials followed by a capitalized name  
                          ; of any length  
*&[*?][@[ ] ]          ; any number of capitalized words or initials  
{##}:{##}:{##}         ; for time
```

---

## Using format specifications to control output

Just as you can use pictures in an ACCEPT command to determine how values are input, you can use format specifications of the FORMAT() function (fully described in the *PAL Reference*) to determine how they are output. You can use format specifications to

- control numeric precision
- justify and align data
- change case of letters
- specify the form in which dates appear
- suppress leading blanks of values when they are displayed or printed

Within the FORMAT() function, you can combine formats of different types by separating the specifications with commas. For example, "W6, AL" specifies left alignment within a width of 6. You can also combine editing formats following a single E ("E\$C" for a floating dollar sign with commas every three digits).

Table 5-3 Format specifications

Format type	Specification	Data types applied to *
Width	Wn allowable width	all
	Wn.m width and decimal places	N,\$
Alignment	AL left justify	all
	AR right justify	all
	AC center within width	all
Case	CU make uppercase	all
	CL make lowercase	all
	CC initial caps only	all
Edit	E\$ floating dollar sign	N,S,\$
	EC use separators between every 3 whole digits	N,S,\$
	EZ print leading zeros	N,S,\$
	EB blanks for leading zeros	N,S,\$
	E* "*" for leading zeros	N,S,\$
	EI international format	N,S,\$
	ES print in scientific notation	N,S,\$

Format type	Specification	Data types applied to *	
Sign	S+	print leading + or - sign	N,S,\$
	S-	print leading - sign	N,S,\$
	SP	print negatives in ( )	N,S,\$
	SD	print DB or CR notation	N,S,\$
	SC	print CR after negatives	N,S,\$
Date	D1	use mm/dd/yy format	D
	D2	use Month dd,yyyy format	D
	D3	use mm/dd format	D
	D4	use mm/yy format	D
	D5	use dd-Mon-yy format	D
	D6	use Mon yy format	D
	D7	use dd-Mon-yyyy format	D
	D8	use mm/dd/yyyy format	D
	D9	use dd.mm.yy format	D
	D10	use dd/mm/yy format	D
	D11	use yy-mm-dd format	D
	D12	use yy.mm.dd format	D
	D13	use dd/mm/yyyy format	D
Logical	LY	use Yes/No for True/False	L
	LO	use On/Off for True/False	L

\* Data types: N = number, S = short number, \$ = currency, D = date, and L = logical

## Width

Width (W) specifications control the total number of characters a displayed or printed value can have. Width values can be between 1 and 255. If you don't specify a width, the entire data value is output.

If the formatted expression is a number, you can also control the number of digits to the right of the decimal point. The total width must include space for the entire value, including

- all digits to the left and right of the decimal point
- the decimal point itself
- the sign (whether or not shown)
- any other characters, such as whole number separators and dollar signs

The number of decimal places cannot exceed 15.



If the length of a number or date value exceeds the format width, a string of asterisks is output. If the width specified isn't enough for the number of decimal places in the expression, the resulting value is rounded. If the length of an alphanumeric or logical value exceeds the format width, the value is truncated. If there isn't enough width specified for the number of decimal places in the expression, the number is rounded.

Here are some examples of width specifications:

```
? FORMAT("w6","This is a test") ; outputs This i
? FORMAT("w6",1234567) ; outputs *****
? FORMAT("w1", (5=5)) ; returns True, outputs T
? FORMAT("w9.2",1234.567) ; outputs 1234.57
```

---

## Alignment

Alignment specifications adjust the placement of an output value within its format width. Thus, when specifying alignment, you must specify a width as well; if you don't, or if the width is equal to the length of the value, alignment will have no effect.

If alignment is not specified, strings and logical values are aligned to the left, numbers and dates to the right.

Here are some examples of alignment specifications:

```
? FORMAT("w20,ac","This is") ; outputs This is
? FORMAT("w20,ac","The Title") ; outputs The Title
? FORMAT("w20,ac","Of the Book") ; outputs Of the Book
? FORMAT("w20,al",123456) ; outputs 123456
? FORMAT("w20,ar",123456) ; outputs 123456
```

---

## Case

Case specifications control the way values are capitalized. For example, the initial capitalization option CC capitalizes the first letter of each word; a word, in this case, is defined as a string of characters up to but not including the next non-letter character.

Here are some examples of case specifications:

```
? FORMAT("cu","the quick brown fox") ; outputs THE QUICK BROWN FOX
? FORMAT("cl","JUMPS OVER THE LAZY") ; outputs jumps over the lazy
? FORMAT("cc","DOG.") ; outputs DOG.
? FORMAT("cc","widgets'r us " + "too") ; outputs Widgets'R Us Too
```

---

## Edit

Edit specifications control the way numbers are shown. You can combine several edit specifications under a single E prefix, although only one of EZ, EB, and E\* can be used. So, if you wanted both a \$ sign and commas to separate whole digits in a number, you would use the format specification "E\$C". Although in this situation the input data type could be currency (\$), numeric (N), or short (S), you could only place the output in an alphanumeric field because of the dollar sign (\$) and comma.

Here are some examples of edit specifications:

```
x = 34567.89
? FORMAT("w10.2, e$c", x) ; outputs $34,567.89
? FORMAT("w10.2, e$ci", x) ; outputs $34.567,89
? FORMAT("w13.2, e$c", x) ; outputs $34,567.89
? FORMAT("w14.2, e$cb, al", x) ; outputs $ 34,567.89
? FORMAT("w15.2, e$cz, al", x) ; outputs $000034,567.89
? FORMAT("w15.2, e$c*, al", x) ; outputs $***34,567.89
```

The last option is often used for writing checks, to protect against unauthorized insertion of digits into a number. When you include an edit specification, use a width specification in the same FORMAT() call.

You may want to use a sign specification (described in the next section) as well as an edit specification to format numbers.

---

## Sign

Sign specifications determine how positive and negative values are distinguished. Sign specifications apply only to numbers, and you can use only one at a time.

Here are some examples of sign specifications:

```
x = -3456.12
? FORMAT("w8.2, s+", x) ; outputs -3456.12
? FORMAT("w11.2, e$c, sc", x) ; outputs $3,456.12CR
? FORMAT("w14.2, e$c*, sp", x) ; outputs ($***3,456.12)
? FORMAT("w13.2, e$c*, s+", x) ; outputs $-***3,456.12
? FORMAT("w14.2, e$c*, sd", x) ; outputs $***3,456.12CR
```

DB (debit) and CR (credit) are used primarily in accounting applications.

You can use sign and edit specifications together. The last three examples above show the order in which sign and edit elements are output:

1. leading parenthesis, if specified
2. \$ sign, if specified
3. + or - sign, if specified
4. fill characters (blanks, zeros, or \*), if necessary and specified
5. the number itself
6. DB or CR, if specified
7. closing parenthesis, if specified

---

## Date

Date specifications output date values in any of the Paradox date formats illustrated in Table 5-3. If you use both a width and date specification, the width should be wide enough to fit the date format; otherwise, Paradox displays a string of asterisks.

Here are some examples of date specifications:

```
? FORMAT("d2", 8/21/1989)      ; outputs August 21, 1989
? FORMAT("d7", 8/21/1989)      ; outputs 21-Aug-1989
? FORMAT("d11", 8/21/1989)     ; outputs 89-08-21
? FORMAT("d7,w5", 8/21/1989)   ; outputs *****
```

Formats d9, d10, d11, d12, and d13 are not supported in Paradox versions 1.0 or 1.1.

---

## Logical

Logical specifications substitute the logical values Yes/No or On/Off for the default values of True/False. Logical specifications apply only to logical values.

For example,

```
? FORMAT("LY", (5 = 5))      ; outputs Yes
```



# Procedures

You can supplement PAL's built-in commands and functions and streamline your applications by defining your own procedures and procedure libraries. This chapter explains

- what procedures are and why you should use them
- how to define and call a procedure, and how to release it from memory
- how procedures use variables
- what closed procedures are and how to use them
- what procedure libraries are and how to create, store and use them

---

## What is a procedure?

A *procedure* is a set of PAL commands that you construct to perform a task; in this sense, a procedure is like a script. However, procedures differ from scripts in several important ways, and these differences translate into direct programming and performance benefits:

- *Power.* Like PAL functions, a procedure can accept arguments and return values. You can construct a procedure to take no arguments or a number of arguments. You can also construct procedures simply to perform actions. This lets you extend the power of PAL in precisely the way you need to.
- *Structure.* Procedures provide more structure than scripts because the flow of information into and out of a procedure is more strictly controlled. If you use procedures correctly, they are easier to understand, test, and debug than scripts.

For instance, you can declare variables *private* to a procedure. When you declare a variable private to a procedure, the variable is treated as a separate variable from one with the same name in

the script that called the procedure (a *global* variable). Because you can isolate the use of a procedure's variable, it's easier to prevent it from causing unwanted side effects.

You can store procedures in procedure libraries. When you need the procedure, you just call it. You can keep most of an application's code in a single, self-contained procedure library.

- *Readability.* Procedures names can be up to 32 characters in length. You can name a procedure in a way that describes its function, such as *DisplaySplashScreen()*, instead of being limited to the 8 characters in a script name.

---

## Defining procedures

A procedure cannot be used until it has been defined with the PROC...ENDPROC statements. Your script can call a defined procedure as often as necessary. You must place all your procedure definitions at the start of a script, or you must store procedures in libraries and read them from the library as needed (libraries are explained later in this chapter).

A procedure definition starts with the PROC command and ends with the ENDPROC keyword. The first line of a procedure definition, called the *header*, consists of

- the keyword PROC
- optionally, the keyword CLOSED
- the name of the procedure
- a parenthesized, comma-separated list of parameters

The parentheses are mandatory, even if the procedure takes no arguments.

The commands between the header and the ENDPROC keyword make up the *body* of the procedure. The commands in this section must be valid PAL statements or calls to functions, or other procedures. For instance,

```
PROC Hypotenuse(x,y)
  w = x * x
  z = y * y
  RETURN SQRT(w + z)
ENDPROC
```

The procedure is defined until

- all script play ends.

- ❑ You use the `RELEASE PROCS` command (described below) to release it from memory.
- ❑ It is defined within a closed procedure and the closed procedure terminates.

Since procedures always start as scripts, you don't have to set out to write procedures directly. When you're new to PAL, you should start by writing your application as a script or set of scripts. As you gain more experience with procedures, you'll find it advantageous to use procedures right from the start.

There are two stages in application development where you'll want to convert script commands into procedures:

- ❑ As you begin to develop your application, you might find small, simple tasks or *subroutines* that your application performs repeatedly, such as checking to make sure the printer is ready each time the application prints a report. Rather than duplicate the code in these subroutines each time, write them once, debug them, and get them working. Then define them as procedures. These procedure modules are a toolkit that you can rely on as you continue to develop the application.
- ❑ Otherwise, you can ignore procedures until later in the application development process. Write the script and get it working before attempting to convert sections of your code into procedures. Once you have a working application, look for large blocks, or *modules*, that you can categorize into procedures.

Once you have a script or script segment that performs an operation, defining it as a procedure is simple. Just put the script lines between the `PROC` command (to begin a procedure definition) and the keyword `ENDPROC` (to end the definition):

```
PROC ProcName(Parameter list) ; parentheses enclose parameter list
                               ; this is where the body of the
                               ; script goes
ENDPROC                       ; ends procedure definition
```

Let's say you have a segment of code that deletes the last record from a table and finds the number of records left in the table.

```
; DelLast1.sc
; deletes the last record in Customer table and returns number of records

COEDIT "Customer"
END
DEL
DO_IT!
MESSAGE "There are ", RECNO(), " records in the customer table."
SLEEP 2000
MESSAGE ""
```

If you want to repeat this operation in other places in your program, make the segment into a procedure and call it as necessary. Just

separate out the four lines from COEDIT to DO\_IT!, name the procedure, and call the procedure when you want to execute those four lines of code.

The next example shows *DelLast1* after it has been made into a procedure.

```
; DelLast2.sc
; procedure to delete the last record in Customer table

PROC DelLast()                ; PROC begins procedure definition.
                                ; DelLast is the name of the procedure.
                                ; Parentheses following procedure name
                                ; are mandatory.

    COEDIT "Customer"
    END
    DEL
    DO_IT!
ENDPROC                        ; ENDPROC ends the procedure definition.

DelLast()                      ; DelLast() is a standalone call.
                                ; When a procedure is called this way,
                                ; it's a standalone call.

MESSAGE "There are ", RECNO(), " records in the Customer table."
SLEEP 2000
MESSAGE ""
```

---

## Calling procedures

There are two ways to call a defined procedure. You can make it a standalone call, like this:

```
DelLast("Customer")
```

Or, if the procedure returns a value, you can call it in any context that uses an expression, just as you can specify a function in any context that uses an expression. The returned value of the procedure is substituted for the expression.

```
x = DelLast("Customer") ; DelLast("Customer") evaluates to
                        ; an expression
```

You can call procedures from a script or from within another procedure.

---

## Input to procedures: Parameters and arguments

When you define a procedure, you can set it up so that it accepts values of some sort on which to act. A *parameter* is the variable into which an argument is passed. You list parameters inside the parentheses that follow the procedure name. Consider another version of *DelLast*:

```
; DelLast3.sc
; procedure to delete the last record in a specified table
; the procedure definition indicates that an argument is required when
; the DelLast procedure is called.
```



```

PROC DelLast(Tbl)           ; Tbl is called a formal parameter to the
                           ; DelLast procedure
    COEDIT Tbl             ; Tbl is merely a variable assigned a value
                           ; when the procedure is called
    END
    DEL
    DO_IT!
ENDPROC

DelLast("Customer")       ; Customer is the argument that is passed to
                           ; DelLast's Tbl parameter

MESSAGE "There are ", RECNO(), " records in the Customer table."
SLEEP 2000
MESSAGE ""

```

"Customer" is the argument passed to Tbl

The preceding example script defines a procedure called *DelLast* with one parameter variable called *Tbl*. It assumes the value of *Tbl* is the name of an existing table. The script calls *DelLast* and, inside the parentheses, supplies a table name as an argument. The procedure substitutes the value of the argument ("Customer") every time it sees the parameter variable *Tbl*. Specifically, the procedure goes into CoEdit mode for the *Customer* table, moves to the last record, deletes the last record, and posts the edit.

Suppose you don't know the name of the table that you want to pass as an argument to *DelLast*. Instead, you want to prompt the user for a name, store it in a variable, and pass the variable as an argument to *DelLast*. Here's how:

```

; DelLast4.sc
; Deletes last record in a specified table. Script passes a variable
; instead of a constant to the procedure.

PROC DelLast(Tbl)           ; define procedure the same as last example
    COEDIT Tbl
    END
    DEL
    DO_IT!
ENDPROC

; create a window to get input from user
WINDOW CREATE @3,5 WIDTH 46 HEIGHT 5 TO InputWindow

@1,1 ?? "Table to delete last record from: "
ECHO NORMAL                 ; show the window
ACCEPT "A8" TO TableName    ; assign user's input to a variable
WINDOW CLOSE
ECHO OFF                     ; hide workspace action

DelLast(TableName)         ; now call the procedure, using the variable
                           ; as the argument to DelLast procedure

                           ; now we can use the input variable in
                           ; the following message

MESSAGE "There are ", RECNO(), " records in ", TableName, " table."
SLEEP 2000
MESSAGE ""

```

In this example, the script prompts the user for a table name, stores the user's response in the variable *TableName*, and then passes the *TableName* variable as an argument to *DelLast*. *DelLast* accepts the argument *TableName* and stores it in the *Tbl* parameter.

When you call a procedure, the actual values of the arguments are assigned to the formal parameter variables given in the procedure definition. This method of passing arguments is called *pass by value*. If your script passes a fixed array or a dynamic array as an argument to a parameter, the array is *passed by reference*, not by value. See the "Variables and procedures" section of this chapter for an explanation of array arguments.

You can pass valid PAL expressions as arguments to a procedure. The expression cannot include a PAL command. A passed expression can include one or more of the following elements:

- constants
- variables
- function calls
- the return value of a procedure call
- field values
- one or more single elements of an array
- fixed arrays
- dynamic arrays

The following example shows how you may use function calls as arguments to procedures.

```
PROC DelLast(Tbl)           ; define DelLast procedure with parameter
  COEDIT Tbl
  END
  DEL
  DO_IT!
ENDPROC

ECHO NORMAL                ; show workspace action
VIEW "Customer"           ; place customer table on workspace
DelLast(TABLE())          ; this call to the DelLast uses the name of
                          ; the current table as the procedure's argument

MESSAGE "There are ", RECNO(), " records in the ", TABLE(), " table."
SLEEP 2000
MESSAGE ""
```

The following example shows how you would pass the return value of one procedure as an argument to another procedure:

```
PROC GetTableName()        ; define procedure without a parameter

WINDOW CREATE @3,5 WIDTH 46 HEIGHT 5 TO InputWindow
@1,1 ?? "Table to delete last record from: "
```

```

ECHO NORMAL                ; show the window
ACCEPT "A8" TO TableName
WINDOW CLOSE
ECHO OFF

RETURN TableName          ; send value of TableName back to calling
                          ; script or procedure
ENDPROC

PROC DeLast(Tb1)          ; define DeLast procedure with parameter
COEDIT Tb1
END
DEL
DO_IT!
ENDPROC

DeLast(GetTableName())   ; This call to the DeLast procedure calls
                          ; the GetTableName procedure, which returns a
                          ; value. The value of GetTableName is passed
                          ; as an argument to the DeLast proc.

MESSAGE "There are ", RECNO(), " records in the ", TableName, " table."
SLEEP 2000
MESSAGE ""

```

Return value of GetTableName is passed as argument to DeLast

The following method for calling procedures is preferred over the method used in the last example.

```

TabName = GetTableName()  ; assign GetTableName's return value to a
                          ; variable

IF ISTANCE(TabName)       ; check to see if the table exists
  THEN DeLast(TabName)    ; then use the variable as argument to DeLast
ELSE MESSAGE "Sorry, ", TabName, " does not exist..."
  SLEEP 2000
  MESSAGE ""
ENDIF

```

## Output from procedures: Return values

You can cause a procedure to return a value to the script or procedure that called it. Put a RETURN command at the end of the procedure, right before ENDPROC. (You can actually put a RETURN command anywhere in a procedure, but it terminates the procedure when it executes. This is useful in some situations, but not in the examples that follow.)

RETURN makes the procedure store a piece of information when it terminates. To work directly with the information, you must call the procedure as part of an expression. For example, you can modify *DeLast* so that it returns the number of records in a table.

```

; DeLast5.sc
; The procedure in this example deletes the last record
; in a table and returns the number of records in the table.

PROC DeLast(Tb1)
COEDIT Tb1
END
DEL
DO_IT!
CLEARIMAGE                ; remove table from workspace

```

```

        RETURN NRECORDS(Tb1)          ; return number of records in the table
    ENDPROC

NewSize = DeLast("Orders")          ; assign value that DeLast returns to
                                       ; NewSize variable

MESSAGE "Orders table now has ", NewSize, " records."
SLEEP 2000
MESSAGE ""

```

When a procedure explicitly returns a value, you can call the procedure in an expression where you want to use the value. The last part of *DeLast5* could be modified as follows:

```

MESSAGE "Orders table now has ", DeLast("Orders"), " records."
SLEEP 2000
MESSAGE ""

```

This example shows how you might use a procedure that returns a logical value:

```

; Pwordtst.sc
; This procedure checks for a valid password and returns a logical
; value (true or false).

PROC GetPassword()
    WINDOW CREATE FLOATING @3,5 WIDTH 30 HEIGHT 5 TO InputWindow
    @1,1 ?? "Enter your password: "
    ACCEPT "A10" TO UserPass          ; get input from user
    RETURN UserPass = "SwordFish"    ; now return True if user entered
                                       ; SwordFish and return False if not
ENDPROC

GoodPass = GetPassword()            ; assign value procedure returns to
                                       ; the GoodPass variable
IF GoodPass                          ; this command is the same as
                                       ; "IF GoodPass = True"
    THEN Main()                      ; password OK so continue application
    ELSE EXIT                        ; exit if password is incorrect
ENDIF

```

---

### ***A note on Retval***

When a procedure returns a value with the RETURN command, Paradox assigns the returned value to the special system variable *Retval*. The script in the following example accomplishes exactly the same thing as *DeLast5*.

```

; DeLast6.sc
; The procedure in this example deletes the last record
; in a table and returns the number of records in the table,
; but uses retval for displaying the message.

PROC DeLast(Tb1)
    COEDIT Tb1
    END
    DEL
    DO_IT!
    CLEARIMAGE
    RETURN NRECORDS(Tb1)          ; return number of records in the table
ENDPROC

```

```

DelLast("Orders")           ; standalone call to procedure
MESSAGE "Orders table now has ", Retval, " records."
SLEEP 2000
MESSAGE ""

```

Paradox always assigns *Retval* the return value of the procedure that was most recently executed. If you think you will ever want to call another procedure or perform an action that changes *Retval* between a procedure and its use of *Retval*, then you should assign the return value of the procedure to an explicit variable. For instance, all of the locking commands, as well as ACCEPT, LOCATE, and WAIT, change the value of *Retval*. (See "System variables," in Chapter 3 for more details.)

## Variables and procedures

You've seen how to pass values to procedures via procedure parameters and how to get a value from the procedure via the RETURN command. But you still need to know the relationship between variable *x* in a script and variable *x* in a procedure.

Usually, any script variable or array is considered a *global variable*. Although there are two exceptions to this rule (discussed later in this chapter), you can think of this as a general rule. Any script or procedure can use or change a global variable. In the following example, the *TableName* variable is given a value in the script, then the script calls *DelLast*, and *DelLast* uses the global variable *TableName*:

```

PROC DelLast()                ; DelLast does not accept an argument
  COEDIT TableName           ; TableName can be used because it's global
END
DEL
DO_IT!
ENDPROC

TableName = "Orders"         ; TableName is a global variable
DelLast()

```

Procedures not only use global variables, they can change the values of global variables. In the next example, *TableName* is set to "Orders", the procedure deletes the last record in *Orders*, then changes the value of *TableName* to "Customer".

```

PROC DelLast()
  COEDIT TableName
END
DEL
DO_IT!
  TableName = "Customer"      ; reassign the global variable!
ENDPROC

TableName = "Orders"         ; make first assignment to TableName
MESSAGE "Global variable TableName = ", TableName
SLEEP 2000:
DelLast()

```

```
MESSAGE "Global TableName has been changed to : ", TableName
SLEEP 2000
MESSAGE ""
```

In small applications, global variables are easy to keep track of and can be an advantage. Large applications become unmanageable, however, and must be broken down into smaller modules. Each module should make its variables private to hide them from other modules.

A variable is *private* to a procedure when it is hidden from the calling script or procedure. This means you can change the value of private variable  $x$  inside a procedure and the value of another variable  $x$  in the calling script or procedure will not change.

Within a procedure, a variable or single element of an array is private if it is

- a formal parameter to a procedure
- a variable explicitly declared as private
- declared within a closed procedure

Fixed and dynamic arrays are handled differently than variables or single elements of arrays. Variables and single elements of arrays, when passed to a procedure, are *passed by value*. This means that the procedure works on a “copy” of the variable or element, but leaves the original alone. Entire arrays, however, are *passed by reference* to a procedure. This means that the procedure parameter *refers* to the actual array. Changes made to the array inside the procedure are happening to the global array.

Closed procedures have special rules for handling variables. See “Closed procedures” later in this chapter for details.

---

## Dynamic scoping

The *scope* of a variable is the area within which that variable has meaning. The scope of a global variable is the entire application (closed procedures are exceptions). The scope of a private variable is the procedure that declares it private and any procedures or scripts called by that procedure. In other words, the scope of a procedure’s private variable is the same as the scope of the procedure itself. For instance, if procedure  $P$  declares a variable  $v$  private, then  $v$  has meaning within procedure  $P$ . Variable  $v$  also has meaning within any script or procedure called by  $P$ , because procedures and scripts called by  $P$  are within  $P$ ’s scope. This process is called *dynamic scoping* because the scoping rules are determined dynamically as procedures and scripts invoke each other. (See Example 6-1.)

---

## Formal parameter variables

Let's start with a new example procedure called *Future*, which takes a date and a number of weeks as arguments. *Future* returns the date exactly the number of weeks away from the date.

```
PROC Future(Date, Weeks)
    Weeks = Weeks * 7
    RETURN (Date + Weeks)
ENDPROC

WeeksAhead = 4
MESSAGE "Result of call to Future is ", Future(TODAY(), WeeksAhead)
SLEEP 3000
MESSAGE "WeeksAhead is still ", WeeksAhead
SLEEP 3000
MESSAGE ""
```

---

## Single variable arguments

When *Future* is called by the MESSAGE command, PAL evaluates TODAY() and assigns its value to *Date* (the first parameter of the *Future* procedure). It passes the value for *WeeksAhead* to *Weeks* (the second parameter of the *Future* procedure); *Weeks* is then changed to 28 (which actually makes it days) and is then added to *Date* to calculate the future date. The value for *Weeks* starts at 4 and then increases to 28; but this change has no effect on *WeeksAhead*. *WeeksAhead* keeps the value of 4.

In a script without a procedure, the same passing of values takes this form:

```
WeeksAhead = 4
Weeks = WeeksAhead
Weeks = Weeks * 7
MESSAGE "Weeks is ", Weeks, ", but WeeksAhead is ", WeeksAhead
SLEEP 2000
MESSAGE ""
```

It's clear in the previous script that once *Weeks* takes the value of *WeeksAhead*, the two variables are no longer related. But procedures actually go one step farther than this. If a script variable is passed to a procedure and that procedure has a formal parameter with the same name, the formal parameter variable acts independently of the script variable. For example,

```
PROC Future(Date, Weeks)
    Weeks = Weeks * 7
    MESSAGE "Inside procedure, Weeks is now ", Weeks
    SLEEP 2000
    MESSAGE ""
    RETURN (Date + Weeks)
ENDPROC

Weeks = 4 ; notice that the script variable is now called
           ; Weeks, the same name as the procedure's
           ; formal parameter

MESSAGE "Future date is ", Future(TODAY(), Weeks)
SLEEP 2000
MESSAGE "Outside procedure, Weeks is still ", Weeks
```

```
SLEEP 2000
MESSAGE ""
```

Since *Date* and *Weeks* are formal parameters of the *Future* procedure, and because they are single variables (not arrays), their lifetime is restricted to the execution of the procedure—when the call to *Future* returns, these formal parameters are forgotten. For this reason, formal parameters like *Date* and *Weeks* are private to *Future*.

---

## Array arguments

Fixed and dynamic arrays are not handled the same as single variables when the arrays are passed to a procedure as formal parameter arguments. Single variables are passed by value; both fixed and dynamic arrays are passed by reference. For example, here's a procedure definition that accepts an array as a formal parameter and changes the elements of that array.

```
PROC ToSpanish(ArrayName)
  WINDOW CREATE @9,23 WIDTH 27 HEIGHT 7 TO ProcWindow
  ArrayName[1] = "Uno"           ; change value of array elements
  ArrayName[2] = "Dos"
  ? "Inside procedure : "
  ? "NumWords[1] = ", ArrayName[1]
  ? "NumWords[2] = ", ArrayName[2]
ENDPROC

ECHO NORMAL
ARRAY NumWords[2]
NumWords[1] = "One"           ; assign initial value to array elements
NumWords[2] = "Two"

WINDOW CREATE @2,2 WIDTH 27 HEIGHT 7 TO BeforeWindow
? "Before procedure call : "
? "NumWords[1] = ", NumWords[1]
? "NumWords[2] = ", NumWords[2]

ToSpanish(NumWords)           ; pass name of array to procedure

WINDOW CREATE @16,43 WIDTH 27 HEIGHT 7 TO AfterWindow
? "After procedure call : "
? "NumWords[1] = ", NumWords[1] ; is now "Uno"
? "NumWords[2] = ", NumWords[2] ; is now "Dos"
```

Note that the array *NumWords* changed as a result of the procedure.

---

## Variables declared private

If you want to make variables or arrays private to a procedure, declare them private with the `PRIVATE` clause. Here's a procedure that uses the Pythagorean theorem to calculate the length of the hypotenuse of a triangle. The `PRIVATE` clause makes *w* and *z* private to *Hypotenuse*, so they do not alter the values of the global variables *w* and *z*.

```
PROC Hypotenuse(x,y)
  PRIVATE w, z
  w = x * x           ; assignments to w and z will not affect
  z = y * y           ; their global values
  RETURN (SQRT(w + z))
ENDPROC
```



```

w = 2 ; assign global variables
x = 4
y = 5
z = 3
WINDOW CREATE TO DisplayWindow
? Hypotenuse(x,y) ; displays 6.403124237432
? "x = ", x, " y = ", y ; displays 4 5
? "w = ", w, " z = ", z ; displays 2 3

```

Arrays can be declared private in the same way as variables: Name the array in the PRIVATE clause as if it were a simple variable, then declare the array in the usual manner in the body of the procedure.

The next example declares a dynamic array and uses the array to set window attributes. The procedure appears to reassign the elements of the dynamic array, but since it is declared private, the original values remain intact when control returns to the script.

```

PROC SetNewAtts(WinHandle)
  PRIVATE WinAtts ; declare dynamic array private
  DYNARRAY WinAtts[]
  WinAtts["HASFRAME"] = True ; assign elements
  WinAtts["HASSHADOW"] = True
  WINDOW SETATTRIBUTES WinHandle FROM WinAtts ; set attributes of window
  ; from the dynamic array
ENDPROC

DYNARRAY WinAtts[] ; declare a dynamic array
WinAtts["HASFRAME"] = False ; initialize array elements
WinAtts["HASSHADOW"] = False

VIEW "Customer"
ECHO NORMAL
WINDOW HANDLE CURRENT TO CustWindow
WINDOW SETATTRIBUTES CustWindow FROM WinAtts ; set attributes from the array
SLEEP 3000
SetNewAtts(CustWindow) ; call procedure
SLEEP 3000
WINDOW SETATTRIBUTES CustWindow FROM WinAtts ; reset attributes from array

```

### Example 6-1 Dynamic scoping

This script shows how Paradox dynamically scopes variables. The script first assigns a value to *x*. Because the assignment is not made within the scope of any procedure that has explicitly declared *x* private, that *x* is a global variable.

The script then calls procedure *Proc1*, which declares a private *x*. Once this *x* is declared private, it is undefined; you must therefore assign a value to *x* before *x* can be referenced within the scope of *Proc1* (the scope of *Proc1* is within procedure *Proc1*, or within any script or procedure called by *Proc1*).

*Proc1* assigns a value to *x* and displays *x*. *Proc1* then calls *Proc2* (this execution of *Proc2* is within the scope of *Proc1* and can thus use a variable that *Proc1* declares private). *Proc2* displays *x*, then calls *ChangeX*.

*ChangeX* declares *x* private, assigns a value to *x*, and displays *x*. Even though *ChangeX* is called within the scope of *Proc1*, *ChangeX* can manipulate *x* and have no effect either on *Proc1*'s private *x* or on the global *x* because *ChangeX* declares *x* private to itself. The script then calls *Proc2*.

Since the second call to *Proc2* is not within the scope of *Proc1*, the *x* that *Proc2* uses is the global variable *x*.

```

; scope.sc
; script to demonstrate dynamic scoping

PROC ChangeX()
    PRIVATE x                ; x is private, so this proc will
    x = 999                  ; not change x outside of this proc's
    ? x, "(x from ChangeX)" ; scope. Because ChangeX calls
ENDPROC                    ; no other procs or scripts, the scope of
                           ; ChangeX's private variables is
                           ; limited to ChangeX.

PROC Proc1()
    PRIVATE x                ; Since x is declared private, it is currently
                           ; undefined. Any use of it at this point will
                           ; cause an error.

    x = 2                    ; x is now defined and can be used within scope
                           ; of Proc1 proc

    ? x, "(private x from Proc1)"
    Proc2()
; Since Proc2 is called by Proc1, it is in Proc1's
; scope. Any variables private to Proc1 can be
; used by Proc2.

ENDPROC

PROC Proc2()
    ? x, "(x from Proc2)"    ; takes x from calling proc or script
    ChangeX()                ; call ChangeX to see if it changes x
    ? x, "(x from Proc2, after call to ChangeX)"
ENDPROC

WINDOW CREATE FLOATING TO DisplayWindow

x = 1                        ; this value of x is global
? x, "(global x)"           ; show current value of x
Proc1()                     ; call procedure Proc1 (declares a private x),
                           ; which then calls Proc2 (uses x from Proc1)
? x, "(global x after call to Proc1)"
Proc2()                     ; Call Proc2. Since Proc2 is called outside
                           ; of the scope of Proc1, Proc1's private
                           ; variables have no meaning during this
                           ; execution of Proc2. Proc2 therefore uses the
                           ; global value of x.
? x, "(global x after call to Proc2)"

c = GETCHAR()                ; any keypress returns to Paradox

```

The preceding script prints this information to the current canvas:

```

1 (global x)
2 (private x from Proc1)
2 (x from Proc2)
999 (x from ChangeX)
2 (x from Proc2, after call to ChangeX)
1 (global x after call to Proc1)
1 (x from Proc2)
999 (x from ChangeX)
1 (x from Proc2, after call to ChangeX)
1 (global x after call to Proc2)

```

---

## Closed procedures

Closed procedures let you create application modules that are mostly self-contained. They also make it easy to take advantage of PAL's built-in memory management capabilities and therefore reduce the need for you to explicitly manage memory resources in your applications. These two attributes make them quite useful in structuring large PAL applications. For such applications, it is advantageous to encapsulate the top-level and major first-level modules into closed procedures.

The principal difference between closed procedures and other PAL procedures is that a closed procedure establishes a self-contained unit. When a closed procedure returns or otherwise terminates, all of the procedures, variables, and arrays called or used by the procedure are removed from memory. Thus, you can define variables, arrays, and other procedures within a closed procedure, but when the closed procedure terminates, all of these definitions and the resources they occupy are released back to the central memory pool.

Because closed procedures automatically release information that would otherwise never be released from memory, they are preferred to using the `RELEASE PROCS` and `RELEASE VARS` commands. However, closed procedures terminate more slowly than other PAL procedures because of the memory maintenance they perform.

The `USEVARS` option lets you include a list of variables to make available to the closed procedure. Variables imported with `USEVARS` are not automatically released when the closed procedure terminates.

When you use closed procedures, you should be aware of the following:

- ❑ Unless you explicitly import variables into a closed procedure through the `USEVARS` keyword, the procedure knows only about variables defined within it.
- ❑ All of the rules that apply to the scope of variables for the entire PAL environment apply internally within each closed procedure.
- ❑ When a closed procedure begins, Paradox saves the current state of the application's procedures and variables. When the closed procedure terminates, Paradox restores the procedure and variable state that existed before the closed procedure was called. If the closed procedure changed the values of any global variables imported into the closed procedure with the `USEVARS` option, the new values are retained when the closed procedure terminates.
- ❑ You can nest closed procedures a maximum of 255 levels deep.

To define a closed procedure, simply add the keyword CLOSED to the header of the procedure. For example,

```
PROC CLOSED Hypotenuse()
  USEVARS x,y                ; global variables x,y are available to
                              ; closed procedure with use of USEVARS
  w = x * x                  ; no need to declare w and z as private -
  z = y * y                  ; their global values will not be affected
  RETURN (SQRT(w + z))
ENDPROC
```

For optimal performance, use closed procedures to structure the main modules of your applications because they automatically return memory used for procedures and variables. Here is the recommended way to use closed procedures to structure large applications:

- The top-level procedure for the application should be encapsulated into a closed procedure.
- Each major module called by the top level should be encapsulated into a closed procedure.
- If a module itself contains several major components, each of these can also be enclosed by a closed procedure; because you can have as many as 255 nested closed procedures, you are unlikely to exceed the maximum.
- Within each major module, regular procedures should be used to structure the code that does the actual work.

The following example shows the definition of a top-level closed procedure:

```
PROC CLOSED TopLevel() ; define closed procedure
  Count = 0            ; initializing variables that will be used
  Flag = False         ; by each module through USEVARS
  WHILE True
    SHOWMENU
      "ViewData" : "View Customers, Orders, and Products",
      "OrderEntry" : "Enter new orders",
      "Edit" : "Change Customer and Products tables",
      "Reports" : "Print a summary report",
      "Exit" : "Exit the application"
    TO Choice

    SWITCH
      CASE Choice = "ViewData" : Browse() ; each of these four
      CASE Choice = "OrderEntry" : OrdEntry() ; major modules is
      CASE Choice = "Edit" : Editprod() ; itself a closed
      CASE Choice = "Reports" : RptProc() ; procedure
                                      ; when the user 'quits'
      CASE Choice = "Exit" : QUIT ; to Paradox, all
                                      ; variables and procs
                                      ; will be released
    ENDSWITCH
  ENDWHILE
ENDPROC
```

Because each closed procedure in the example is a self-contained unit, when control is passed back to *TopLevel*, all resources used by the procedure will automatically be released and restored to the central memory pool. Therefore, when the closed procedure containing the next module of the application is called, it will have the same resources as if it were the first procedure called when the application was originally started.

Note the use of the global variables *Count* and *Flag* in *TopLevel*. The values of these variables can be read by each of the closed procedures through the USEVARS option; when the closed procedure ends, the current value of the variables will be available to all of the other modules called from the *TopLevel* procedure.

Since *TopLevel* is itself a closed procedure, when the application terminates, it will leave no trace of itself because all resources used by *TopLevel* are returned to the central memory pool.

In versions of Paradox prior to 4.0, you were required to read closed procedures from libraries. This restriction has been removed from Paradox 4.0.

Within the realm of a closed procedure, all rules that apply to other Paradox procedures pertain. For example, all of the scoping rules for variables that apply to Paradox as a whole apply within the realm of each closed procedure; all of the swapping rules described in Chapter 21 apply to closed procedures.

---

## Procedures and libraries

Procedure libraries are special files that store one or more procedure definitions in a parsed form; this makes it very quick to both read them into memory and to execute them. Paradox's procedure library capability gives you several benefits:

- Procedures stored in libraries are maintained in parsed form and cannot be read by other programmers—only by PAL. Procedure libraries are an ideal vehicle for protecting your applications.
- Procedure libraries help you consolidate the numerous scripts that might comprise a large application into a more manageable package—a single procedure library together with a small “driver” script. Once a procedure library is built, the original scripts containing the procedure definitions are not needed to run the application, only to debug it.
- Procedure libraries provide an ideal vehicle for creating and storing sets of personal productivity macros. Your *Init* script can

include SETKEY commands that call procedures in the autoloading library, causing macros to be read in and executed upon demand.

Procedures in libraries are made available to your application by a special autoloading feature that allows automatic loading of a called procedure stored in one or more designated libraries.

---

## Creating procedure libraries

Here is an overview of the steps needed to create and use a procedure library:

1. Use the CREATELIB command to create a procedure library that will hold a group of procedures. When you create a library, Paradox creates a file with the extension .LIB having space for 50 procedures.

To store more than 50 procedures in a single library, use the optional SIZE keyword and specify a library size of up to 640. Because the file-size overhead for library files is approximately 2K per 50 procedures allocated (this is independent of whether any procedures are actually stored in the library), it's not wise to specify more space in a library than you will realistically use.

2. Execute the definitions of one or more procedures and use the WRITELIB command to write the definitions into the library.
3. Create a "driver" script to launch your application. The driver script should assign your procedure libraries to the *Autolib* variable and then call the main procedure. See "Autoloading procedures" later in this chapter.)

Procedures in libraries assigned to the *Autolib* variable are available to Paradox. You do not have to use the READLIB command to read procedure libraries into memory, as you may have done in versions of Paradox prior to 4.0.

The CREATELIB command creates a procedure library. A library must be explicitly created with CREATELIB before any procedures can be stored in it. CREATELIB takes as an argument a string expression that designates the name of the library. PAL gives library files a .LIB file extension, so

```
CREATELIB "Myprocs"
```

creates a file called MYPROCS.LIB.

This script creates a library called Ordprocs, defines the *DellLast* and *Future* procedures, and writes them to the library.

```
CREATELIB "Ordprocs"           ; must create a library before
                               ; writing to it
PROC DellLast(Tbl)           ; define DellLast procedure
  COEDIT Tbl
END
```

```

DEL
DO IT!
RETURN NRECORDS(Tb1)
ENDPROC

PROC Future(Date, Weeks)           ; define Future procedure
  Weeks = Weeks * 7
  RETURN (Date + Weeks)
ENDPROC

WRITELIB "Ordprocs" DelLast, Future ; write the two procedures to library
                                       ; notice that this is one of the few
                                       ; times that parentheses () are not
                                       ; used when referring to a procedure

```

Typically, a separate script (frequently called a “make” script) is used to create the procedure library and write procedures to it. This script contains a CREATELIB command, followed by one or more PROC...ENDPROC definitions, and finally a WRITELIB command to write the procedures into the library. After the library is written, this script containing the procedure definitions is no longer needed except for debugging. Example 6-3 shows one such script.

---

## Storing procedures in libraries

Once you have used CREATELIB to create a library, you use WRITELIB to store procedures in it. You can store up to 640 procedures in a single library. WRITELIB takes as arguments the name of the procedure library and a list of procedures to be written out to it. Thus, to use WRITELIB, the procedures you want to store must currently be in memory (defined) and the library must exist.

You can use WRITELIB repeatedly to store additional procedures in a library. Writing a procedure with the same name as one already written to the library causes the old definition to be lost, but the library still contains all the old data. If you want to recover the space in a library, you should recreate the library and write the procedure to it again.

**Note** Do not give two procedures the same name, even if you’re storing them in different libraries. *Autolib* does not recognize multiple procedures with the same name.

---

## Using procedures from libraries

Once a procedure is stored in a library, it can be used in a script by loading it explicitly with the READLIB command or by letting PAL load it automatically with an *Autolib* definition. Loading a procedure from a library has exactly the same effect as defining it with the PROC command. Example 6-2 shows how you would convert an application from scripts to procedures called from a library.

---

## Autoloading procedures

You can use PAL’s autoloading library feature to read in procedures automatically. When a procedure is called that has not yet been defined, PAL looks for a library file named PARADOX.LIB in the

current working directory. If this library is present, PAL searches it for the specified procedure and, if found, reads it into memory. If the library is not present or the procedure is not found, a script error indicates that the called procedure has not been defined.

You can designate another library as an autoload library by assigning its name to the system variable *Autolib*. For example,

```
Autolib = "sam"
```

causes PAL to search SAM.LIB instead of PARADOX.LIB.

You can also set up an autoload library search path to designate any number of libraries to be autoload libraries. For example,

```
Autolib = "sam,joe,mylib"
```

tells Paradox to search for procedures in three libraries: SAM.LIB, JOE.LIB, and MYLIB.LIB. You can include as many libraries as will fit in a 255-character string. The order in which the libraries are listed in the path determines the order in which they are searched at run time. If the string assigned to *Autolib* contains the names of several libraries, separate the library names with commas, but *don't* include any spaces.

You can use different sets of autoload libraries in the same application by assigning different values to *Autolib* at the appropriate times. In addition, by declaring *Autolib* as a private variable within a procedure, you can change the autoload libraries just for the duration of the call to that procedure.

---

### **Explicitly loading procedures**

Although autoloading procedures is preferred, you can use the READLIB command to explicitly load procedures from a library into memory. READLIB names the procedure library to be read along with a list of procedures to be loaded. READLIB IMMEDIATE tells Paradox to load all the procedures into memory at the same time, although they will still be swapped out if Paradox needs room. This is useful if you have a set of procedures that need to be used together. You can use READLIB to read in as many or as few procedures as are needed at a given time. If there isn't enough memory to load all of the procedures named in the command, READLIB triggers a script error.

This script loads and calls the *DelLast* procedure from the Ordprocs library created in the last example:

```
READLIB "Ordprocs" DelLast  
MESSAGE "There are ", DelLast("Orders"), " records left in Orders"
```



## Example 6-2 Using procedure libraries

---

Suppose you're developing an order-entry application in which the top-level menu looks like the following example:

```
WHILE True

  SHOWMENU
    "OrderEntry" : "Enter new orders",
    "View"       : "Look at orders",
    "Reports"    : "Print reports",
    "Exit"       : "Quit the application"
  TO choice

  SWITCH
    CASE choice = "OrderEntry" : PLAY "Addat"
    CASE choice = "View"       : PLAY "Looksee"
    CASE choice = "Reports"    : PLAY "Prnreprt"
    CASE choice = "Exit"       : QUIT

  ENDSWITCH

ENDWHILE
```

In a script without procedures, the *Addat* script would contain the commands needed to handle the processing of new orders. Now suppose the *Addat* routines are modular enough to be made into separate procedures; you could (as described in the next section) write them into a library called *Addat*.

If you did, instead of playing a script, you would need to call only the procedures. The following method takes advantage of the enhanced autoload library and procedure memory management capabilities in version 2.0 and later. Simply put all the procedures in the autoload library and call them directly in your script.

```
AutoLib = "EntryApp"          : EntryApp library contains major procedurized
                               : modules used in the Order Entry application

WHILE True
  SHOWMENU
    "OrderEntry" : "Enter new orders",
    "View"       : "View the Orders table",
    "Reports"    : "Produce reports",
    "Quit"       : "Quit the application"
  TO Choice

  SWITCH
    CASE Choice = "OrderEntry" : OrdEntry() ; these procedures are
    CASE Choice = "View"       : ViewOrd()  ; loaded automatically
    CASE Choice = "Reports"    : ReportSum() ; from the EntryApp library
    CASE Choice = "Quit"      : QUIT

  ENDSWITCH

ENDWHILE
```

Using this method, only the top-level procedure is called. There's no need to explicitly read it into memory; PAL handles this task automatically. Similarly, there's no need to explicitly release the procedure from memory because PAL automatically frees memory resources as needed by swapping procedure definitions out of memory.

## Listing the contents of a library

The INFOLIB command creates the temporary *List* table that shows the procedures stored in a specified library. The table includes an approximation of the number of bytes each procedure occupies in memory. For example,

```
INFOLIB "Ordprocs"
```

displays a *List* table containing a record for each procedure stored in the Ordprocs library.

### Example 6-3 Creating a procedure library

This script creates a library called Birthday, defines the procedures in the *NextBDay* script, and writes them to the library.

```
CREATELIB "Birthday" ; create the library

PROC DaysAway(Date,YrsMore)
  NextDate = DATEVAL(STRVAL(MONTH(Date)) + "/" +
    STRVAL(DAY(Date)) + "/" +
    STRVAL(YEAR(TODAY()) + YrsMore))

  IF (NextDate = "Error")
    THEN RETURN 367
    ELSE RETURN NextDate - TODAY()
  ENDIF
ENDPROC

PROC NextBDay(Date)
  DaysAway = DaysAway(Date, 0)
  IF DaysAway < 0
    THEN DaysAway = DaysAway(Date, 1)
    ELSE
      IF DaysAway > 366
        THEN RETURN "You do not have a birthday this year."
      ENDIF
    ENDIF
  IF DaysAway = 0
    THEN RETURN "Happy Birthday!"
    ELSE RETURN "There are " + STRVAL(DaysAway) +
      " days left until your birthday."
  ENDIF
ENDPROC

WRITELIB "Birthday" DaysAway, NextBDay

Autolib = "Birthday" ; Birthday library will be searched
; upon subsequent procedure calls

WINDOW CREATE @3,5 WIDTH 35 HEIGHT 5 TO InputWindow
@1,1 ?? "Enter your birthdate: "
ECHO NORMAL
ACCEPT "D" TO UserBday
WINDOW CLOSE
RETURN NextBDay(UserBday) ; call the procedure - will be
; automatically loaded from
; Birthday library
```

### Example 6-4 Reading and using library procedures

---

The following script autoloads procedures from the Birthday library created in Example 6-3:

```
Autolib = "Birthday"

WINDOW CREATE @3,5 WIDTH 33 HEIGHT 5 TO InputWindow
@ 10,10 ?? "What is your birthdate: "
ECHO NORMAL
ACCEPT "D" TO Bday
WINDOW CLOSE
ECHO OFF
RETURN (NextBDay(Bday))
```

---

#### More on WRITELIB

Writing a procedure with the same name as one already stored in the library causes the old definition to be lost, without a warning message. The disk space occupied in the library by the old definition cannot be recovered, nor can procedures be removed from a library. This means that reducing the size of a procedure and rewriting it to the library does not reduce the size of the library; to reduce the size of the library, you should rebuild it. Thus, instead of creating one gigantic, catch-all library, it is better to create several smaller libraries containing only the procedures you need for a given operation.

If you do need to rebuild a library, you'll have to create a new one from scratch (with CREATELIB), write all of the current procedure definitions into the new library, and then delete the old one. If you create procedure modules and then build them into an application, you'll end up periodically recreating your libraries in order to reclaim the space taken by obsolete or changed procedures.

**Note** The limitations of libraries are no problem if you organize your work as shown in Example 6-3 and Example 6-4; and use one or more scripts to create libraries and then write procedures to them, with one small top-level script.

---

#### Using libraries from previous versions of Paradox

Paradox 4.0 does not load or execute library files created by earlier versions of Paradox. Libraries created in versions of Paradox prior to 4.0 must be converted before they can be used in Paradox 4.0. Attempting to use an obsolete .LIB file will cause a script error.

PAL provides the FILEVERSION() function and the CONVERTLIB command to help you work with these older libraries. The FILEVERSION() function returns the number of the Paradox version that created the library, as follows:

```
RETURN FILEVERSION("MYLIB.LIB")
```

When you use FILEVERSION(), supply the name of the library within quotation marks with a .LIB extension. The library name must be a valid library file or a script error will result.

Use the CONVERTLIB command to convert a Paradox 3.0 or 3.5 library file to a Paradox 4.0 library as follows:

```
CONVERTLIB "C:\LIBRARY\MYLIBOLD" "C:\LIBRARY\MYLIBNEW"
```

Supply the full path and name of the old and new library files within quotation marks and without the .LIB extension. The old and new library names cannot be identical.

---

## **Debugging procedures stored in libraries**

In Paradox 4.0, you can debug procedures in libraries even if the original source is not available. If the source code for a procedure is unavailable, you will see the message "Source unavailable" when you enter the Debugger; however, you still have access to the Debugger menu *Alt-F10*.

Because a procedure is stored in a library in parsed form, the source code cannot be changed unless the script, including the original procedure definition, is accessible to the Debugger. The full path of the script is stored in the library along with the procedure so PAL can find this definition.

If you want to be able to change the source code of a procedure that is stored in a library, make sure the procedure definition scripts are available in the drive and directory from which they were originally written into the library. The Debugger will look for the script in its original location; not necessarily in the current directory.

Once debugged, it is up to you to save a new, corrected version of the procedure into the library.

# Special topics

This chapter covers how to

- password protect your scripts
- use the special scripts *Init* and *Instant*
- trap and process errors
- run external programs from Paradox

---

## Password protection

You can use Paradox's Tools | More | Protect command to password-protect (encrypt) scripts as well as tables. Protecting a script prevents those who do not know the password from examining or modifying its text, not from executing it.

As with tables, when a script is protected using Tools | More | Protect, it is encrypted immediately. Any subsequent attempt to edit or debug the protected file will cause Paradox to request the script's password; this is done only *once* per session (until the password is cleared). Encryption prevents a protected script from being edited even in an external text editor and from being interrupted with *Ctrl-Break*. Scripts encrypted by Paradox 4.0 will not run in previous versions of Paradox.

When you change the protection status of a script, the pre-parsed .SC2 file is deleted. The next time the script is played, a new, appropriately encrypted or decrypted .SC2 file is generated.

For efficiency, you can group together a number of tables and scripts by giving them all the same password. Once the password for the group has been presented, all of the tables and scripts in the group can be accessed without further challenge.

As an application developer, here's what you need to know about protecting your tables and scripts:

- ❑ You can use either `Tools | More | Protect` or the abbreviated `PROTECT` command to protect a table with an master password. Generally, you'll use `Tools | More | Protect` to protect the tables you include with the application, and `PROTECT` to protect any tables that the application creates.

You can also use `Tools | More | Protect` to define auxiliary passwords that give tables more elaborate protection. Auxiliary passwords let you define different levels of rights to the table, its fields, and its family, based on which password is entered by the user. See Chapter 17 of the *User's Guide* for details on master and auxiliary passwords and encryption.

- ❑ Scripts can be protected only with `Tools | More | Protect` and not with `PROTECT`.
- ❑ Before using a protected table in a script, you must supply a valid master or auxiliary password either in a `PASSWORD` command or as a response to Paradox's password prompt (see Example 7-1).
- ❑ `PASSWORD` presents a password; it does not encrypt a table. `UNPASSWORD` retracts a password, reprotecting a table after access; it does not unencrypt a table.
- ❑ If you use a protected table in a script, you should protect the script as well to safeguard the password. If you don't, anyone could examine the script and discover the password. Also, users don't need the password to play the script.
- ❑ If you want users to provide the password to use a protected table instead of supplying it for them, use a variable in the `PASSWORD` command and have the user input the value for the variable (see Example 7-1).
- ❑ If you want the user to provide a password to play a script, you'll need to create your own prompt and comparison method (see Example 7-1).
- ❑ At the end of your application, make sure you issue `UNPASSWORD` to retract passwords and leave tables protected, then issue a `RESET` to remove any tables from the workspace and from the table buffers in memory. See the `UNPASSWORD` and `RESET` commands in the *PAL Reference* for further information. If you don't `UNPASSWORD` the tables, they remain unprotected during the rest of the Paradox session.
- ❑ In terms of performance, there is no penalty for using a protected object. It takes time to encrypt an object, but once encrypted, the object is manipulated just as fast.

On a network, where multiple users are using the same table, using auxiliary passwords to determine who can access what is almost a requirement of creating a successful application. For example, you might assign different auxiliary passwords to

- ❑ data entry clerks, to grant only Entry table rights (so they can enter new data, modify data in non-keyed fields, and view all data) and no family rights (so they aren't allowed to change forms, reports, validity checks, and so on)
- ❑ data entry supervisors, to grant InsDel table rights (so they can correct typographic errors made by a clerk)
- ❑ application supervisors, to grant All rights to tables, families, and fields (so they can maintain and modify the application)

In a network environment, presenting a valid password to access an object provides access only for the session at the workstation on which the password was presented. Users at all other workstations are precluded from using the object unless they too present a valid password.

The auxiliary password system is fully described in the section on Tools|More|Protect in Chapter 17 of the *User's Guide*.

Here's a summary of the protection and password process:

1. Use PROTECT to ask Paradox to encrypt the table. Assign a password in the same statement.
2. To use the table, present the password with the PASSWORD command. As Paradox uses the table it unencrypts blocks of it into the table buffers. If any part of a table is unencrypted on a workstation, Paradox considers that table as available to the workstation.
3. When the application is finished with the table, retract the password with UNPASSWORD.
4. Remove the table from the workspace with CLEARALL.
5. Remove the unencrypted blocks from the table buffers in memory with SAVETABLES or RESET.

#### Example 7-1 Protecting scripts and tables

---

Suppose you've developed an application to manage a department of secret agents. Since the information is highly sensitive, you've already used Tools|More|Protect to password-protect the script (*MiÓ*) and tables (*Staff* and *Agents*). In addition, to prevent unauthorized access, you want users to give a password to run the script. That's enough clearance to view or modify the *Staff* table, but they need to supply an additional password to view or modify the *Agents* table. Here is how you might implement this protection scheme:

```

; define procedure to get password
PROC CLOSED GetPass(GoodPass)

    WINDOW CREATE @3,5 WIDTH 40 HEIGHT 5 TO InputWindow
    @1,1 ?? "Enter password: "           ; prompt for password
    ECHO NORMAL                          ; show window and prompt
    ECHO OFF                              ; don't show user's typein
    FOR Attempt FROM 1 TO 3              ; allow 3 attempts
        ACCEPT "A15"
        REQUIRED                          ; require a response
        TO UserPass                      ; assign entry to variable
        IF (UserPass = GoodPass)        ; check for correct password
            THEN WINDOW CLOSE
            RETURN True
        ELSE MESSAGE "Incorrect password!"
            SLEEP 1500
            MESSAGE ""
        ENDIF
    ENDFOR
    MESSAGE "You obviously don't know the password..."
    SLEEP 3000
    EXIT                                 ; exit to DOS after 3 tries
ENDPROC

GetPass("GoldFinger")                  ; call procedure to get password to run
                                        ; this script

WHILE True
    SHOWMENU
    "Staff" : "View the Staff table",
    "Agents" : "View the Agents table",
    "Quit" : "Quit the application"
    TO Choice

    SWITCH
        CASE Choice = "Staff" :
            PASSWORD "largo"           ; present password for Staff table
            VIEW "Staff"
            WAIT TABLE
            PROMPT "Browse the table, press [F2] when done"
            UNTIL "F2"
            UNPASSWORD "largo"         ; revoke password for Staff table
            CLEARIMAGE                 ; now table is protected again
            RESET
        CASE Choice = "Agents" :
            GoodPass = "MoneyPenny"    ; this is the expected password
            ClearUser = GetPass(GoodPass) ; call procedure to get password
            IF ClearUser                ; if user entered correct password
                THEN
                    PASSWORD GoodPass
                    VIEW "Agents"
                    WAIT TABLE
                    PROMPT "Browse the table, press [F2] when done"
                    UNTIL "F2"
                    UNPASSWORD GoodPass ; revoke password for Agents table
                    CLEARIMAGE         ; now table is protected again
                    RESET
                ENDIF
            OTHERWISE : QUIT
    ENDSWITCH
ENDWHILE

```

Routine asks for password to access protected table

GetPass procedure protects script from being played without authorization

PASSWORD command accesses protected table

UNPASSWORD reprotects table after access

The above procedure creates a canvas window and uses ACCEPT to solicit a password from the user. You can also create a dialog box with SHOWDIALOG that uses ACCEPT with the HIDDEN keyword to solicit a



password that will not appear onscreen when the user enters it. See Chapter 15 for complete information about using dialog boxes to interact with the user.

Password protection is not a trivial matter. Tables and scripts should be protected from novice users (who might damage important information) and from those who should not change them. Maintaining protection while allowing users to perform useful and relevant work takes care and planning. Keep scripts and tables protected when possible, and unprotect them only when necessary.

See Tools | More | Protect in Chapter 17 of the *User's Guide* for a general discussion of the protection schemes available and how they affect users' access to data.

If you are developing multiuser applications for a network environment, you have access to additional features that allow you to protect data from use or changes by others. For further information, see Chapter 23 of this manual for details on implementing multiuser protection schemes.

---

## Special scripts

Paradox creates and uses several special scripts in your private directory, including

- *Init*, played by Paradox at the start of each session (if found)
- *Instant*, for scripts recorded with Instant Script Record *Alt-F3* and played back with Instant Script Play *Alt-F4*
- *Value*, created and executed when you use PAL Menu | Value to evaluate an expression and then automatically deleted (see Chapter 8)
- *Mini*, created and executed when you use PAL Menu | MiniScript to execute a single-line script and then automatically deleted (see Chapter 8)
- *Savevars*, created when you use SAVEVARS to save the current values of variables and array elements (see the *PAL Reference*)
- *Execute*, created and executed when you use the EXECUTE command to process a text string and then automatically deleted (see the *PAL Reference*)

You should not use these names for scripts you create, unless you deliberately want to create a special effect. For example, you might want to create your own *Savevars* script to preload a set of trial variable and array values for testing purposes.

---

## Init

When you start Paradox, it looks in your private directory for a script called *Init*. If found, it is played before the Main menu is displayed and before any script named on the DOS command line is executed.

You can use *Init* to customize Paradox according to your needs. For example, you can include SETKEY commands to customize the keyboard (see Chapter 20 on keyboard macros), and MESSAGE or RETURN commands to display messages at the bottom of the screen.

---

### Example 7-2 A sample *Init* script

Here's a typical *Init* script that changes the default directory, sets up three keyboard macros, and notes the day or date.

```
SETDIR "c:\paradox\gift"

SETKEY -83 PLAY "SafeDel"      ; Del key plays the SafeDel script
SETKEY 47 MENU                 ; forward slash invokes the menu
SETKEY "F11" PLAY "PhoneLst"  ; Shift-F1 executes PhoneLst
IF DOW(TODAY()) = "Fri"
    THEN RETURN "TGIF!"
    ELSE RETURN "Today is " + TODAY()
ENDIF
```

---

## Instant

*Instant* scripts are especially useful

- ❑ to create ad hoc keyboard macros that automate repetitive key sequences
- ❑ to prototype operations on the fly (later you can use the Editor to merge the *Instant* script into your application)
- ❑ as a productivity tool to quickly capture and replay important sequences of commands

Use Instant Script Record *Alt-F3* to record a special script named *Instant*, and Instant Script Play *Alt-F4* to play it back. Each time you use Instant Script Record *Alt-F3*, any existing script named INSTANT.SC is deleted. To save an instant script for future use, you can rename it using Menu {Tools}{Rename}{Script}.

---

## Errors

---

### Types of errors

There are three types of errors that can prevent your scripts and applications from running properly:

- ❑ *Syntax errors* result from commands that are not well-formed or that PAL does not recognize. They are typically caused by misspellings, or missing or misplaced elements in expressions, as in:

```

VIEW "Orders"           ; misspelling of "View"
billday = DOW(TODAY() + 30 ; missing right parenthesis

```

- *Run-time errors* arise from commands that are syntactically valid, but which for one reason or another cannot be carried out. A run-time error occurs if you try to view a table that does not exist; you'll also get a run-time error if you try to perform an operation in an incorrect sequence or in an incorrect mode. For example,

```

VIEW "Orders"
SCAN
  [Price] = [Price] * 1.6
ENDSCAN

```

produces a run-time error, since Paradox must be in Edit or CoEdit mode in order to assign values to fields within images. You'll also get a run-time error if Paradox does not have sufficient resources to carry out an operation.

- *Logic errors* result from code that is syntactically correct, but that is based on faulty algorithms or incorrectly implemented control structures, such as endless loops or calculations that return incorrect results. Logic errors can sometimes produce run-time errors; for example, a script that calls itself recursively can eventually use up all available memory.

PAL will trap and report syntax errors and run-time errors. However, it cannot detect logic errors as such. These you will have to discover yourself by carefully testing your programs to make sure they are producing the expected results.

---

## Error processing

Since errors invariably happen in programs, your applications should expect the unexpected. Paradox provides several facilities to trap errors, find out what went wrong, and take action to deal with the problem gracefully in your script:

- The `ERRORCODE()` function returns a code number describing the most recent run-time error or error condition (see Table 7-1).
- The `ERRORMESSAGE()` function returns the text of the message Paradox presented at the last run-time error or error condition.
- The `ERRORUSER()` function is useful in a multiuser context to determine the name of the user who locked a resource.
- The `ERRORINFO` command creates a dynamic array with information about the latest run-time error. This dynamic array contains elements with tags that represent error attributes.
- The system variable `Errorproc` lets you assign the name of an error-handling procedure. The designated error procedure is executed when a run-time error occurs. By default, `Errorproc` is unassigned (no error procedure).

Table 7-1 Paradox error codes

<b>Code</b>	<b>Meaning</b>
0	No error (for example, record was not changed or key was found)
<b>File or directory errors</b>	
1	Drive not ready file errors
2	Directory not found
3*	Table in use by another user
4*	Full lock placed on table by another user
5	File not found
6	File corrupted
7	Index file corrupted
8	Object version mismatch
9*	Record locked by another user
10	Directory in use by another user
11*	Directory is private directory of another user
12	No access to directory at operating system level
13	Index inconsistent with sort order
14	Multuser access denied
15	PARADOX.NET file conflict
<b>General script errors</b>	
20	Invalid context for operation
21	Insufficient password rights
22	Table is write protected
23	Invalid field value
24	Obsolete procedure library
25	Insufficient image rights
26	Invalid PAL context
27	Operation not completed
28	Too many nested closed procedures
29	Table is remote
<b>Argument errors</b>	
30	Data type mismatch
31	Argument out of range
32	Wrong number of arguments
33	Invalid argument
34	Variable or procedure not assigned
35	Invalid menu command
36	Missing command parameter

<b>Code</b>	<b>Meaning</b>
37	Nested complex expression
<b>Resource errors</b>	
40**	Not enough memory to complete operation
41	Not enough disk space to complete operation
42**	Not enough stack space to complete operation
43	Printer not ready
44	Low memory warning
<b>Record-oriented operation errors</b>	
50**	Record was deleted by another user
51**	Record was changed by another user
52**	Record was inserted by another user
53**	Record with that key already exists
54**	Record or table was not locked
55**	Record is already locked by you
56**	Lookup key not found
57**	Record and group lock dependencies on this table lock
<b>Multi-table operation errors</b>	
60	Referential integrity check
61	Invalid multi-table form
62**	Form locked
63	Link locked
64	Group locks applied to table with different key
65	Group of records locked
66	Cannot read file due to incompatible file version

\* For these codes, ERRORUSER() returns the name of the user who locked the resource.

\*\* These codes do not invoke a designated error procedure.

---

## Error procedures

PAL's error procedure feature enables your script to retain control in the event that a run-time error occurs in the course of script play. While error procedures are useful for all types of applications, they are particularly useful to handle errors in multiuser processes.

The error procedure feature lets you define a procedure that will be invoked in the event of a run-time error. The procedure can determine the nature of the run-time error, and at least in some cases, take corrective action. After executing the body of the procedure, control returns to the point in the script at which the error occurred;

you can retry the statement that triggered the error, skip to the statement following it, or invoke the Script Break menu.

Provided one is defined, PAL automatically executes the error-handling procedure whenever a run-time error is encountered. Some errors do *not* invoke an error procedure (see Table 7-1). You can have different error procedures for different parts of your program if you want. Simply reassign *Errorproc* in each module.

To define an error procedure, set the special system variable *Errorproc* to a string giving the name of the procedure. The error procedure itself must not take any argument and is defined in the same way as any other procedure. An error procedure can be stored in and called from a regular or autoload library. Inside the error procedure, you can use the `ERRORCODE()` and `ERRORMESSAGE()` functions to determine the nature of the error. The `ERRORCODE()` function returns an integer code representing the type of error. `ERRORMESSAGE()` returns a string giving an error message.

A better way to determine the nature of the error is to use the `ERRORINFO` command from within the error procedure. `ERRORINFO` creates a dynamic array with information about the latest run-time error, including `CODE`, `USER`, and `MESSAGE` tags whose values are the same as the values returned by the `ERRORCODE()`, `ERRORUSER()`, and `ERRORMESSAGE()` functions.

For example, if a run-time error was produced by your system being out of disk space, the `ERRORINFO CODE` tag (and the `ERRORCODE()` function) has a value of **41** and `ERRORINFO MESSAGE` tag (and the `ERRORMESSAGE()` function) has a value of **Not enough disk space to complete the operation**. See the *PAL Reference* for complete information about the `ERRORINFO` tags.

Some of the errors (those marked with \* in Table 7-1) result from resource conflicts in network applications. For these errors, use the `ERRORUSER()` function to determine the name of the user who currently holds the shared resource.

Your error procedures should return one of the integer values 0, 1, or 2, depending on whether and how you want to resume script execution:

- 0 causes script execution to resume at the statement triggering the error; in other words, the statement is re-executed (equivalent to Step *Ctrl-S* in the Debugger).
- 1 causes the statement triggering the error to be skipped, so that execution resumes at the next statement following the current one (equivalent to Next *Ctrl-N* in the Debugger).
- 2 displays the Script Break menu (which lets you choose Cancel or Debug), just as if there had been no error procedure defined.

Returning any other value (or returning no value) has the same effect as returning 2. You should return a 0 only if the error procedure has taken corrective action; otherwise, the run-time error is immediately retrigged. For unexpected conditions that display only a message, you can use the WINDOW() function to return the message in the message window. WINDOW() returns only system-generated messages.

As a matter of good coding style, don't rely too much on error procedures. Instead, you should avoid script errors by testing for error conditions explicitly (such as drives or printers that are not ready or tables that are locked) using the appropriate PAL commands and functions.

### Example 7-3 Using error procedures

---

This example shows how you might use an error procedure as a safety net in a password-protected application.

```
ErrorProc = "Handler"           ; designate Handler as error procedure

PROC Handler()                 ; define procedure
  PRIVATE ErrorProc            ; in case of errors within Handler
  ERRORINFO TO ErrorData       ; load dynamic array with error information
  IF (ERRORCODE() = 41)        ; if out of disk space
    THEN QUIT ERRORMESSAGE()  ; quit and give user the error message
  ENDIF
  SAVEVARS ALL                 ; otherwise it's safe to write a file
  MESSAGE "Error occurred! Please call Hank at 555-4300"
  QUIT
ENDPROC
```

Note the declaration of *Errorproc* as a private variable of the error procedure. Within *Handler*, *Errorproc* is declared private but unassigned. Thus, if another run-time error occurs during execution of *Handler*, the error procedure mechanism is not reinvoked. This technique avoids recursion and should be used whenever there is a possibility that a run-time error could be triggered inside the error procedure itself. As an alternative, you could also execute

```
RELEASE VARS Errorproc
```

within *Handler* to deassign *Errorproc*. The use of the private declaration has the advantage that it automatically reinstates the error procedure if control returns to the script.

---

## Running external programs from Paradox

You can use the RUN command to run other DOS programs from within a Paradox application. The RUN command loads a second copy of COMMAND.COM into memory. The NOSHELL option does not load a second copy of COMMAND.COM, but you can't use NOSHELL to run a batch file.

You can pass information directly from your application to another program or vice versa through the DOS command line or through file I/O. For example, if the program you are running can write out a *Savevars* script, you can pass parameters from the program to your application when you exit back into Paradox.

See the *PAL Reference* for a complete description of the RUN command.



# PAL facilities

Now that you know the basic components of PAL, it's time to introduce the built-in facilities with which you can create and test PAL scripts. This part of the manual describes the tools that help you prototype and fine-tune your applications.

Part II consists of three chapters:

- Chapter 8, "The PAL menu." The PAL menu is the home base for many of Paradox's programming facilities. Here you'll find tools to record and play scripts, debug scripts, play short miniscripts, and determine the value of PAL expressions.
- Chapter 9, "Creating and playing scripts." This chapter reviews and summarizes all the methods you can use to create and play scripts. It includes suggestions on when to use script recording methods, the PAL Editor, or other ASCII editors in developing applications.
- Chapter 10, "The PAL Debugger." This chapter presents the built-in debugging tool you can use to test your scripts. Since the Debugger is such an important tool in the development process, this chapter includes a short tutorial showing its use.

The information in these chapters is important to anyone developing PAL applications. The better you know the built-in programming facilities, the easier you will find it to create complete applications. You might also want to refresh your memory by rereading Chapter 18 in the *User's Guide*.



# The PAL menu

The PAL menu gives you instant access to PAL from anywhere in Paradox. This chapter explains how to display and leave the PAL menu and use it to

- record and play back scripts
- enter the PAL Debugger
- calculate the value of an expression
- execute short scripts

---

## Displaying the PAL menu

To display the PAL menu, press PAL Menu *Alt-F10*. You can do this at nearly any point in Paradox. Unless you are already recording or debugging a script, the PAL menu contains six choices:

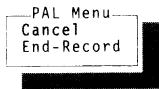


- Play: Plays back a script.
- RepeatPlay: Plays back a script a specified number of times.
- BeginRecord: Begins recording keystrokes in a new script.
- Debug: Debugs a script.
- Value: Calculates the value of an expression or variable.
- MiniScript: Composes and executes a short PAL script.

The PAL menu works just like any Paradox menu. Make choices in the usual way or press *Esc* to return to what you were doing.

If you are debugging a script, pressing PAL Menu *Alt-F10* displays the PAL Debugger menu instead of the PAL menu. The Debugger menu is discussed in Chapter 10.

If you are recording a script when you press PAL Menu *Alt-F10*, you see only two choices:



- ❑ Cancel: Stops recording the script without saving it.
- ❑ End-Record: Finishes and saves the script.

As you'll see in Chapter 9, you can mix and match the PAL Menu `BeginRecord` and `End-Record` commands with `Scripts | BeginRecord` and `Scripts | End-Record`. That is, you can use the PAL menu to finish recording started with `Scripts | BeginRecord`, and vice versa.

---

## Play

Use `Play` to play any script. When you choose `Play`, a dialog box prompts you for the script name; you can type it, or press *Enter* and choose it from the list box. Then Paradox plays the script.

The effect of choosing `Play` from the PAL menu is identical to that of choosing `Scripts | Play` from the Main menu. The only difference is that you can use `Play` from the PAL menu at almost any point in Paradox, without having to return to the Main menu.

For example, if you want to begin playing a script while you are in the Report Designer (where `Scripts | Play` is not a menu command), press PAL Menu *Alt-F10* and choose `Play`.

---

## RepeatPlay

Use `RepeatPlay` to play a script a specified number of times. When you choose `RepeatPlay`, a dialog box first prompts you for the name of the script.

As with `Play`, either type the script name or press *Enter* and choose it from the list box. A Paradox dialog box then asks you to specify the number of times you want to play the script. Type a number, or type the letter **C** to play the script continuously.

The script plays the specified number of times; if you've specified continuous play, the script plays over and over until you press *Ctrl-Break* to cancel it.

---

## BeginRecord

Use `BeginRecord` from the PAL menu to start recording a new script. It has the same effect as `Scripts | BeginRecord Alt-F3`, but has the advantage of being available in almost any context in Paradox. It is therefore useful for prototyping operations for an application by recording sequences of keystrokes from any part of the program.

When you choose `BeginRecord` from the PAL menu, a dialog box asks you to name the new script. If the name you type already exists, you'll have a chance to change the name or replace the existing script.

You can use either `End-Record Alt-F3` or `Scripts | End-Record` to finish recording the script.

For example, if you start recording a script using `Scripts | BeginRecord` and want to end it in the Report Designer, press `PAL Menu Alt-F10` and choose `End-Record`. `End-Record Alt-F3` lets you end script recording at any time within Paradox, even if you don't have access to `Scripts | End-Record` on the Main menu.

---

## Debug

`Debug` gives you access to the PAL Debugger, with which you can test and correct a script by stepping through it one instruction at a time.

When you choose `Debug`, a dialog box prompts you for the name of a script to debug; you can type it, or press `Enter` and choose it from the list box. Then Paradox enters the Debugger.

If the script is encrypted (password-protected), you must supply the correct password before debugging it. The Debugger is described in Chapter 10.

---

## Value

`Value` is a powerful calculator that lets you immediately determine the value of any valid PAL expression.

You can use `Value` to

- calculate arithmetic expressions, such as  $3.14 * 5 * 5$
- determine the current value of variables and arrays, like `Retval`
- invoke PAL functions, such as
  - `PMT(100000, .11/12, 360)`, to calculate the monthly mortgage payment for a \$100,000 loan at 11% per annum over 30 years
  - `DOW(5/12/87)`, to determine a day of the week
  - `TIME()`, to find out what time it is

When you choose `Value`, a dialog box asks you to type the expression you want to evaluate.

You can type up to 175 characters, and use the arrow keys with or without the *Ctrl* key to scroll through and edit the expression. (See the *User's Guide* for details about using the arrow keys when editing.)

You can redisplay and edit the string most recently typed for the Value or MiniScript command. After you've used either command at least once, choose it again, then press *Ctrl-E* at the prompt to redisplay the string. Also, you can add the most recently typed string to the end of a new string: Press *Ctrl-E* after you type the new expression.

When you press *Enter*, PAL calculates and displays the value of the expression in a window in the bottom right corner of the screen. The value remains until the next keystroke or mouse click.

Chapter 3 contains a detailed discussion of the PAL expressions you can use with Value and how they are calculated. Remember that mixed data types in an expression are converted in the result. For example, the mixture of date and number values in the expression `TODAY() + 60` results in a date value (see "Data types" in Chapter 3). You can use this feature to check how mixed data types in an expression will be converted *before* you use it in your script; you can also use it to check the syntax of an expression.

---

## Examining the values of variables and array elements

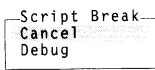
As described in Chapter 3, you can use PAL variables and arrays for temporary storage of data values. You can use any variable name or array element in an expression to be evaluated by Value.

For example, suppose you've received a script error and suspect the problem might involve a variable called *Choice*. Using Value to check its value might tell you that *Choice* is currently unassigned, the wrong data type, or perfectly normal.

---

## Errors in expressions

Value works by first creating and then executing a one-line script called *Value*. Therefore, if the expression contains an error or can't otherwise be evaluated, you'll see the Script Break menu (described in Chapter 9):



- If you already know what's wrong with the expression, choose Cancel and start over.
- Or you can choose Debug to enter the Debugger, where you'll see an error message that explains what went wrong. Press *Ctrl-E* to redisplay the *Value* script dialog box and edit the most recent expression.

One of the most common errors is using Value to determine the value of a variable that hasn't yet been assigned. Using a variable in an expression before it has been assigned a value causes a script error. If you actually meant to enter a MiniScript, press *Ctrl-Q* to quit, then choose MiniScript and press *Ctrl-E* to redisplay the expression.

---

## MiniScript

MiniScript is a quick way to compose and execute a single-line script without having to use the Editor or another script recording method. When you choose MiniScript from the PAL menu, a dialog box asks you to type the script.

You can enter almost any sequence of valid script commands up to a limit of 175 characters. Use the arrow keys with or without the *Ctrl* key to scroll through and edit the miniscript. (See the *User's Guide* for details about using the arrow keys when editing.)

**Note** There are a couple of limitations to building miniscripts. First, a few commands (RETURN, for instance) don't allow another command on the same line and thus cannot be entered except at the end of a miniscript. Second, the PROC command is not allowed in a miniscript.

You can redisplay and edit the string most recently typed for the MiniScript or Value command. After you've used either command at least once, choose it again, then press *Ctrl-E* at the prompt to redisplay the string. Also, you can add the most recently typed string to the end of a new string: Press *Ctrl-E* after you type the new expression.

When you type a miniscript and press *Enter*, PAL creates and executes a temporary script named *Mini*. As with any other script, *Mini* is stacked on top of the script that's currently playing (if any) and becomes the new current script. If an error occurs, debugging begins with the *Mini* script.

Miniscripts have several useful applications in PAL, including

- giving you command-line access to PAL and Paradox
- temporarily patching variables while debugging or testing a script
- creating keyboard macros
- learning how to use PAL

### Example 8-1 A miniscript macro

---

Suppose you want to be able to sort the current table according to the values in the current field by just pressing *Shift-F1*.

Choose MiniScript and type

```
SETKEY "F11" MENU {Image} {OrderTable} ENTER ENTER {Ok}
```

This SETKEY command assigns to *Shift-F1* the command to sort the current table on the current field. Now, when you press *Shift-F1*, the records in the current table are sorted according to the values in whatever field the cursor is in, even if the table is keyed.

---

## Command-line access to PAL and Paradox

Miniscripts are PAL's command line. When you choose MiniScript, you can interactively enter an arbitrary command or sequence of commands that will immediately be interpreted and executed. Through the Miniscript command line, you can use PAL's abbreviated menu commands to directly manipulate Paradox and Paradox objects without going through the menus.

For example, suppose you want to sort the *Customer* table by zip code into a new table called *Ziplist*. You could choose Modify|Sort from the Main menu and then fill out the sort form for *Customer*. Alternatively, you could use the following miniscript:

```
SORT "Customer" ON "Zip" TO "Ziplist"
```

Or, you could use a miniscript to create a secondary index for the zip code field of *Customer*:

```
INDEX MAINTAINED "Customer" ON "Zip"
```

---

## Using miniscripts in testing

While debugging a script, you might use MiniScript to temporarily assign a value to a critical variable, then resume script play. For example, the miniscript

```
x = 3
```

assigns the value 3 to variable *x*. Since PAL variables keep their values throughout a Paradox session, your script can use the variable when it resumes.

Or suppose you're developing an application that consists of a set of small scripts. The script you're working on makes use of a variable called *Choice* that's assigned in another script—which you haven't written yet. To test the script you're writing now, choose MiniScript and assign a test value to *Choice*. Then choose Play and play back the script.

---

## Using miniscripts to set up keyboard macros

Miniscripts are ideal for executing SETKEY commands (described in the *PAL Reference*), which assign a command or sequence of commands to a single key. Once assigned, pressing the key causes the commands, called a *keyboard macro*, to be executed. SETKEY assignments, like variable assignments, last for the length of a Paradox session and are not permanently maintained by Paradox.

If the macro is longer than 175 characters, you can record the commands in a regular script or use the Editor to create a script. Then use a miniscript to attach that script to a key:

```
SETKEY "F11" PLAY "Sortmacr"
```

In this example, *Sortmacr* is the script that contains the commands you want to assign to *Shift-F1*.



Keyboard macros can be extremely helpful to both developers and users of PAL applications. You can store a whole set of macros in a single script to load them all at once. Or you can load them automatically by putting them in your *Init* script. Creating an *Init* script is discussed in detail in Chapter 20.

---

## Using miniscripts to learn PAL

While you're still learning about PAL, MiniScript lets you see the effect of a PAL command without having to use the Editor or other recording facilities to compose a regular script. If you want to see what

```
Menu {Add} {Orders}
```

does, for example, choose MiniScript and type those commands. When you press *Enter*, they'll be executed and you can see the result. In this way, you can experiment with PAL commands interactively.

---

## Screen display by miniscripts

Any use of the screen by a miniscript—or by any script, for that matter—ends as soon as script play ends. A typical miniscript takes so little time to execute that any resulting screen display is visible for only a split second before you return to the Paradox workspace. There are several ways to keep this display visible:

- ❑ Create a canvas window to display the output. Canvas windows (other than floating canvas windows) remain onscreen after a miniscript terminates. See Chapter 11, "Using windows," for more information about canvas windows.
- ❑ Use the `GETCHAR()` function or a `GETEVENT` loop at the end of the miniscript to freeze the screen display until a key is pressed.
- ❑ Use the `SLEEP` command to create an artificial delay while the miniscript is executed.
- ❑ Use `RETURN` or `QUIT` to terminate the script and display a message onscreen.

---

### Example 8-2 Screen display in a miniscript

Suppose you want a miniscript to display the message "Luxury Gifts Department" onscreen. This is what you'd type:

```
ECHO NORMAL CLEAR @12,27 ?? "Luxury Gifts Department"
```

This miniscript would flash "Luxury Gifts Department" so fast you might not even see it. To make the message readable, display it on a canvas window that remains onscreen:

```
WINDOW CREATE TO DisplayWindow ECHO NORMAL @12,27 ?? "Luxury Gifts Department"
```

Because the output is now in a canvas window, it will remain onscreen when the script terminates. Simply close the window when the display is no longer needed.

You could also use the `GETCHAR()` function to pause script execution until a key is pressed:

```
ECHO NORMAL CLEAR @12,27 ?? "Luxury Gifts Department" z=GETCHAR()
```

Press any key to clear the screen when the display is no longer needed.

A `GETEVENT` loop can be used in a way that is similar to `GETCHAR()`:

```
ECHO NORMAL CLEAR @12,27 ?? "Luxury Gifts Department" GETEVENT KEY "ALL" TO Dyn
```

See Chapter 13 for complete information about `GETEVENT`.

Finally, you could append a `SLEEP` command to make PAL pause for two seconds after displaying the message, like this:

```
ECHO NORMAL CLEAR @12,27 ?? "Luxury Gifts Department" SLEEP 2000
```

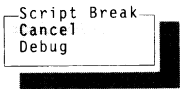
Alternatively, use the `RETURN` command to display a message *after* the miniscript has done its work and finished executing. For example,

```
VIEW "Orders" RETURN "This is the Orders table"
```

displays *Orders* and presents the message *This is the Orders table* in a window in the bottom right corner of the screen. The message remains until you press a key or click the mouse.

---

## Miniscript errors



Script Break  
Cancel  
Debug

If a miniscript contains an error, you'll see the Script Break menu (described more fully in Chapter 9):

- ❑ If you already know what's wrong with the miniscript, choose **Cancel** and start over.
- ❑ Or you can choose **Debug** to enter the Debugger, where you see an error message that explains what went wrong. From the Debugger, press *Ctrl-E* to edit the miniscript and repair it. Otherwise, press **Quit** *Ctrl-Q* to quit the Debugger and start again.

One common error is using `MiniScript` when you meant to use `Value` and vice versa. Remember, `Value` evaluates *expressions* while `MiniScript` executes *commands*. If you make the wrong choice, quit, then choose the other and press *Ctrl-E* to redisplay the most recent `MiniScript`.

---

## Quitting the PAL menu

There are two ways to exit the PAL menu:

- Most of the commands on the PAL menu automatically exit from the menu when they execute. For example, choosing `Debug` takes you out of the PAL menu and into the `Debugger`, while `BeginRecord` leaves the PAL menu and returns you to the `Paradox` workspace in `Record` mode.
- Otherwise, you can exit the PAL menu and return to the `Paradox` workspace any time by pressing `Esc`.



# Creating and playing scripts

So far, almost everything in this book has dealt with the content of PAL scripts—what commands to put in them and how to organize them. This chapter describes PAL's facilities for creating and playing these scripts. It covers

- how to create scripts
- what recorded scripts look like
- how to play scripts
- script errors and interruptions

---

## Creating scripts

There are several ways of creating PAL scripts:

- recording Paradox keystrokes by
  - pressing Instant Script Record *Alt-F3*
  - choosing BeginRecord from the Paradox Scripts menu or the PAL menu
  - choosing QuerySave from the Paradox Scripts menu
- creating scripts from scratch

Recording keystrokes offers the convenience—and limitations—of preserving the actual keystrokes and selections you make as you use Paradox. While useful applications can be created by keystroke recording alone, from the perspective of the programmer, keystroke recording is useful for creating keyboard macros, for creating small pieces of a large application, and for application prototyping.

For creating complete applications, you can write scripts from scratch using the PAL Editor or your own editor. This method gives you

access to all PAL commands. The only drawback is that you must type them yourself.

These script creation choices are described in detail in the following sections.

---

## Recording keystrokes

---

There are four ways to record scripts: Instant Script Record *Alt-F3*, Scripts | BeginRecord, PAL Menu BeginRecord, and Scripts | QuerySave.

---

### Instant Script Record

---

The quickest and easiest way to record a script is to press Instant Script Record *Alt-F3* while in Paradox. Without further prompting, you'll be placed immediately in Record mode (an **R** in the lower right corner of the screen reminds you that you are recording). All your keystrokes and menu choices are recorded in a script called *Instant*. To finish recording the script, press Instant Script Record *Alt-F3* again.

You cannot record mouse events in an instant script. Because users can change the size and shape of a window, recording a mouse event would be deceptive. For example, if you wanted to record a mouse event that clicked the close box of a window, your script would fail when a user moved the window before your script clicked the screen location. For this reason, the mouse is turned off when you are in Record mode.

You cannot start recording an *Instant* script if you are in the midst of recording or playing one already. Because it is a Paradox temporary object, if an *Instant* script already exists, pressing Instant Script Record *Alt-F3* causes Paradox to overwrite the existing script file when you begin recording a new one.

When you start to record an *Instant* script, Paradox won't tell you that you have an existing one; it simply erases it and starts a new one. Thus, if you use this method to capture a sequence of keystrokes that you want to save, be sure you rename the *Instant* script with Tools | Rename | Script or use the Editor to read (merge) it into another script *before* you record another *Instant* script.

**Note** If a private directory is specified, the *Instant* script is always located in the private directory, even though it appears in the list of files in the working directory.

*Instant* scripts are useful primarily

- as temporary macros to repeat keypresses in a single Paradox session, such as

```
Menu {Image} {ColumnSize} Enter Left Left Left Left Left Enter  
; narrows current table column five characters
```

(see Chapter 20 for details on creating more permanent macros)

- to capture a small set of actions that are incorporated into another script, such as

```
Menu {Modify} {CoEdit} {Orders} ; coedit Orders table
```

- to start recording somewhere other than the Main menu

---

### **Scripts|BeginRecord**

Another way to record a script is to choose Scripts|BeginRecord from the Paradox Main menu. A dialog box will prompt you to enter a name for the script.

Once you've chosen BeginRecord and provided a valid name, you'll see an **R** in the lower right corner of the screen, indicating that you are recording. From that point until the end of the recording session, any keystrokes and menu choices you make are recorded in the script.

While you are recording, End-Record replaces BeginRecord on the Scripts menu. This means that you can't begin recording another script until you finish the current one. You can, however, play another script while recording (this is a very convenient way to chain scripts together).

Your keystrokes and menu choices are recorded until you choose Scripts|End-Record from the Main menu. Choosing End-Record completes the recording process and saves the script in a file with the extension .SC. You can also end a script by pressing *Alt-F3* again or by pressing PAL Menu *Alt-F10* and choosing End-Record.

---

### **PAL Menu|BeginRecord**

You can display the PAL menu at any time by pressing PAL Menu *Alt-F10*. Its BeginRecord command works exactly like Scripts|BeginRecord as described previously. To finish recording, choose End-Record from either the Scripts or the PAL menu.

Why would you want to start recording from the PAL menu if it works just like Scripts|BeginRecord? The reason is that Scripts|BeginRecord is available only from the Main menu and not from subsystem menus such as Report, Form, Edit, or CoEdit. Since you can access the PAL menu at any point in Paradox, you can start recording a script in almost any context (except when you're already recording or playing a script).

---

### **Scripts|QuerySave**

Scripts normally consist of representations of sequences of keystrokes you want to replay at a later time. To record a query statement, for instance, you might choose Scripts|BeginRecord, choose Ask from the Main menu, bring up a query form for a table, and enter checkmarks and selection criteria on the query form. The result might look like this:

```
Main {Ask} {Customer} Tab Tab Check "Fischer" Tab Tab Check
```

This example displays a query form for *Customer*, puts a checkmark in the Last Name field, enters a name to search for, tabs over two fields to City, and checks the City field.

However, it is simpler to save a graphic image of the finished query form. This is what QuerySave does. The result is much easier to read and edit than the equivalent sequence of recorded keystrokes.

To record a query using QuerySave, first construct the query interactively by choosing Ask from the Main menu to display the query forms; fill in the query forms as you normally would to create the query statement you want to save. Then choose Scripts | QuerySave from the Main menu and provide a script name when prompted by the dialog box.

All query forms on the workspace are saved in a graphic representation of the query statement. See “Query images” in Chapter 2 for a complete description of all the elements of QuerySave scripts.

When you want to include queries in a longer recorded script, first use QuerySave to record all the queries you want to use. Then, when recording the larger script, use Scripts | Play to display each set of the saved query forms. (Playing a saved query doesn’t execute it, Paradox just puts it on the workspace.) You’ll need to press Do-It! *F2* to execute each query.

You can use parameters in a recorded query by including tilde variables. See the sections on tilde variables in Chapters 3 and 19.

Since query statements are composed of ASCII text, it’s possible to edit their representation by using PAL’s built-in Editor or any other text editor. However, if you are making minor changes to edit the actual query forms on the Paradox workspace:

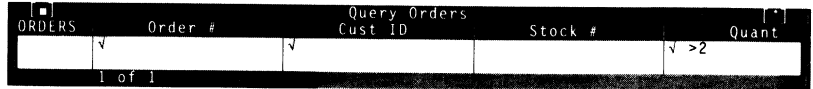
1. Choose Scripts | Play from the Paradox menu and play the script in which the query statement is stored.
2. Once the query forms are displayed on the workspace, edit them interactively to make the necessary changes.
3. Use Scripts | QuerySave to resave the query statement into the script.

This method is preferable to editing the stored ASCII representation because using an editor will not give you the structured support for filling in query forms that is available when query forms are active on the Paradox workspace. For example, when you fill out a query form on the workspace, Paradox checks for errors; the Editor doesn’t.

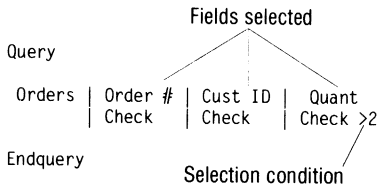


### Example 9-1 A query form and its QuerySave image

Suppose you query the *Orders* table for customers who ordered more than two of some item. Here's the query form:



And here's the image as it's recorded in a QuerySave script.



Remember that the QuerySave script does not include a DO\_IT! command. You must include a DO\_IT! yourself to execute the query when you run the script.

### What recorded scripts look like

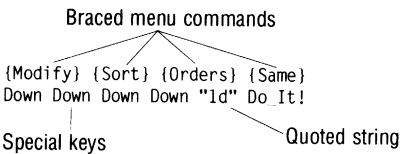
When you use Instant Script Record *Alt-F3*, BeginRecord, or QuerySave, Paradox automatically turns your keystrokes or query forms into a script. Scripts you record in this way can contain four kinds of elements:

- menu choices
- quoted strings
- Paradox special keys
- query images

Example 9-1 and Example 9-2 show scripts that include these elements, and Chapter 2 contains detailed descriptions of them.

### Example 9-2 A sample recorded script

Here's a recorded script that sorts the sample *Orders* table by date, with recent orders first:



If you know Paradox, you can easily understand this script. The first line contains braced menu commands that display a sort form for the *Orders* table. In the second line, the Down key is used to move to the Date field and a quoted string enters the sort specification into the sort form. Then Do\_It! *F2*

carries out the sort. The third line records how the script was ended; it is ignored when the script is played.

---

### **What is recorded**

Although you can type menu commands, key names, and quoted strings directly into a script, it's usually easier to record them. PAL is smart about recording what you do. For example, it will not record:

- ❑ attempts to select an object that doesn't exist
- ❑ use of arrow keys and *Enter* to make menu choices
- ❑ keys used to edit a typed response to a prompt

For example, if you press → twice to highlight Report on the Main menu, then press *Enter*, only the action {Report} appears in your script. Similarly, if you're prompted for a file name and type **journey**, *Backspace* twice, type **al**, and press *Enter*, only the final name "journal" appears in your script. This prevents your scripts from being cluttered with erroneous, extraneous, or confusing commands when you view them.

Menu commands are recorded with initial capital letters, exactly as they appear on the menu. Responses to prompts and quoted strings are recorded exactly as typed. Special keys are recorded by name as shown in Table 2-1 in Chapter 2. Recorded scripts place commands, key presses, and responses in a single line, starting a new line each time the *F10* key is pressed to select the menu. You can edit a recorded script by breaking up lines into smaller blocks and adding comments. Editing a script in this manner makes it easier to understand; however, be careful not to introduce any errors when you are editing.

QuerySave scripts are graphic representations of query forms. See the discussion of Scripts|QuerySave earlier in this chapter for details.

---

### **Editors**

Any text editor that can process straight, uncoded ASCII text can be used to write or edit Paradox scripts. You can use the Editor built into Paradox or your favorite ASCII editor.

---

#### **The Paradox Editor**

Paradox's built-in Editor has all the basic features you need to write and edit scripts. Its main advantage is its integration with the rest of Paradox:

- ❑ You don't have to leave Paradox to edit scripts. It's easy to examine or edit the contents of a script while in Paradox.
- ❑ The Editor is part of PAL's integrated programming environment, which lets you test and correct scripts quickly. For example, with one keypress you can

- go directly from the Debugger to edit the line of code you're debugging
- exit from the Editor and execute the script
- You can use the Editor to edit encrypted scripts as long as the correct password has been supplied during the Paradox editing session.
- The Editor makes it easy to merge one script into another—a feature that is particularly useful for QuerySave and instant scripts.

You can start the Paradox Editor almost any time by pressing *Alt-E*, or selecting Scripts | Editor from the menu bar. Choose Open to open an existing script or New to create a new script.

When it is used to work with scripts the Paradox Editor displays a GO command on its menu bar. Selecting the GO command runs the script in the active window.

The Paradox Editor is described completely in Chapter 18 of the *User's Guide*.

---

### **ASCII text editors**

Every programmer has a favorite editing tool. Despite the advantages of the built-in Editor, you may want to use your own editor to make most of your changes to scripts.

Since scripts are stored as ASCII text files, without embedded control codes or special characters, you can use any editor capable of creating ASCII-only text files to create scripts. Examples of such editors include Borland's Brief and Borland's SideKick Notepad. There are also a number of editors specifically designed to work with Paradox and PAL available from third party vendors.

You can use the Custom Configuration Program to link your own editor into Paradox. This option, which is described in detail in Chapter 15 of *Getting Started*, substitutes your editor whenever the Paradox Editor is called. It gives your editor most of the integration advantages of the Paradox Editor, including editing encrypted scripts. However, depending on the capabilities of your editor, you may not have the seamless integration described previously, such as directly editing the line you're debugging, or exiting and executing the current script in one step.

If you don't link your editor into Paradox, you can still use it to edit scripts, but it will be much less convenient. In particular, you will have to leave Paradox each time you want to modify a script, and you won't be able to edit encrypted scripts.

---

## Summary

Here are some suggestions on when to use each method of creating a script:

- Use the Paradox Editor or your own ASCII editor to
  - make or change an existing script or application
  - enhance a script with PAL commands that cannot be recorded
  - assemble several scripts into one
  - write scripts from scratch
- Use script recording methods to create simple scripts and pieces of more complex applications.
- Use Instant Script Record *Alt-F3* to record repeated actions to replay within a Paradox session, or to capture a script that will be renamed immediately after recording.
- Use Scripts | BeginRecord to record sequences of operations and menu choices that start from the Main menu.
- Use BeginRecord from the PAL menu to record operations that begin when the Main menu is not displayed.
- Use Scripts | QuerySave to record queries.

---

## Playing scripts

The first time a script is executed and any time a script is changed, Paradox loads the script into memory, parses it, and saves this “pre-parsed” version of the script on disk as an .SC2 file. After the .SC2 file is created, a script runs faster because Paradox can use this pre-parsed version of the script. Paradox can temporarily remove a pre-parsed script from memory while it is running, in the same manner that it swaps procedures. For more information about pre-parsed scripts, see Chapter 21.

Once you’ve created a script, you can play it in one of six ways:

- Choose Play from the Paradox Scripts menu.
- Choose Play from the PAL menu.
- Choose RepeatPlay from either the Scripts or the PAL menu.
- Press Instant Script Play *Alt-F4* to play the *Instant* script.
- Name the script on the command line when you start Paradox or Paradox Runtime.

- The *Init* script in the private directory is played automatically when you start Paradox.

---

## Scripts|Play

One way of playing a PAL script is to choose Scripts|Play from the Paradox Main menu. Enter the name of the script you want to play in the dialog box that prompts you. A script you play in this way must start from the Main menu.

Normally, only the end result of the script appears on your screen. If you want to see each action in the script as it is played back, choose ShowPlay instead of Play. See Chapter 18 of the *User's Guide* for details about ShowPlay.

Play appears in the Scripts menu even while you are recording a script. This lets you chain scripts together.

---

## Play from the PAL menu

A second way to play a script is to press PAL Menu *Alt-F10* and choose Play.

This method works almost the same as the Scripts|Play command described previously. There are two differences, however:

- Play from the PAL menu can begin at any point in Paradox (however, the script must operate correctly from the point where you start it).
- Play does not appear on the PAL menu while a script is being recorded.

---

## Scripts|RepeatPlay and RepeatPlay from the PAL menu

You can use the RepeatPlay command on either the Scripts menu or the PAL menu to play a script a specified number of times.

The option you choose usually depends on where you are in Paradox. When you choose RepeatPlay, a dialog box first prompts you for the name of the script; as with Play, you can either type the script name or choose it from the list box. A dialog box then asks you to specify the number of times you want to play the script. Enter a number or type *c* to play the script continuously.

The script will be played the number of times you specified; if you've specified continuous play, the script will be played repeatedly until you press *Ctrl-Break* to cancel it. You'll then get the Script Break menu (described more fully in Chapter 10). At this point, you'll usually want to choose Cancel to end script play.

RepeatPlay is useful when you've recorded a short sequence of keystrokes that you want to repeat several times in succession, such as during a line drawing operation in the Form Designer or Report Designer.

---

## Instant Script Play

Pressing Instant Script Play *Alt-F4* is a quick way of playing the script named *Instant*. You can do this at any point in Paradox, not just in Main mode.

There may be times when you would like to use Instant Script Play *Alt-F4* to play a more complex script than you can construct through keystroke recording alone. In such cases, you can write a script called *Instant* or rename an existing script to *Instant*. Once you've done this, the new *Instant* script plays each time you press Instant Script Play *Alt-F4*. If you rename a script to *Instant* using Tools | Rename | Script, the script is automatically moved to the private directory, if one exists.

---

## Playing scripts from DOS

The last way of playing a script is normally used only for complete applications: Type the name of the script on the DOS command line when you start Paradox. For example, if the script is called *Gift*, type **paradox gift** and press *Enter*.

Scripts played from the DOS prompt must begin in Main mode. If the script contains an EXIT command, you'll be returned to DOS at the end of script play. If the script contains a QUIT command (or no specific final command), you will be left in Paradox when the script finishes. When Paradox starts, it will load settings stored in the PARADOX.CFG file (settings made in the CCP). If these settings name a working directory other than the one containing the application scripts, a script error results.

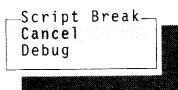
If you are only playing a script, you don't even need Paradox to run it—you can use Paradox Runtime instead. This lets you develop applications for use by others who don't have Paradox.

---

## Script errors and interruptions

Unfortunately, the fact that you've managed to successfully create a script doesn't mean it will always run without problems. The types of errors you or your users may encounter when playing scripts are described in Chapter 7.

When a run-time error or syntax error occurs, you'll see the Script Break pop-up menu:



- Cancel: Stops playing the script and returns to Paradox.
- Debug: Enters the PAL Debugger (see Chapter 10) to find and fix the problem.

Using the Debugger, you view the effect of each instruction in the script to understand where it went wrong. You can then edit the script so that it runs properly the next time. In addition, if the problem is a run-time error, you can temporarily fix the problem by interactively changing the context in which the script is running.

For example, if your script has tried to enter values into a table without being in Edit or CoEdit mode, you can choose Debug from the Script Break menu to enter the Debugger, press Edit *F9*, then press Go *Ctrl-G* to continue playing the script. While this method will get around the problem one time, if you want the script to run correctly in the future, you'll have to correct the code in the script. If the problem is a syntax error, you won't be able to correct the problem interactively; you will first have to edit the code and then start the script over again.

---

## Interrupting script play

Unless a script is encrypted (password-protected), you can interrupt it by pressing *Ctrl-Break*. You'll see the Script Break menu described in the previous section, where you can choose either to cancel or debug it. *Ctrl-Break* may not have an immediate effect in some instances, since PAL will not stop running until it reaches the next statement in the script. You cannot interrupt an encrypted script with *Ctrl-Break* unless the password has previously been presented. See "Password protection" in Chapter 7 for additional information about protected scripts.





# The PAL Debugger

The PAL Debugger is a powerful tool you can use to test your scripts and find errors. With it you can

- ❑ find the cause of an error
- ❑ evaluate expressions interactively
- ❑ check values of variables and arrays and save them in a file
- ❑ insert additional commands during script play
- ❑ determine where you are within nested levels of scripts and procedures
- ❑ return from nested scripts or procedures
- ❑ trace through command execution one command at a time
- ❑ step through an application without tracing into procedures, `PLAY`, `EXECUTE`, or `EXECPROC` commands
- ❑ skip individual commands in a script
- ❑ directly enter the Editor or your own editor to change the script being tested

The Debugger is part of PAL's integrated environment for prototyping, developing, and testing scripts and applications.

This chapter explains how to start the Debugger, work with it, and exit. At the end of the chapter, there is a sample debugging session which illustrates debugging techniques.

---

## Starting the Debugger

There are four ways to start the Debugger:

1. Choose Debug from the Script Break menu, which appears when an error occurs in a script being played. The current command will be the one in which PAL detected the error.

There are two kinds of script errors trapped by PAL:

- Syntax errors, which result from commands that are not syntactically correct, or that PAL does not recognize. They are typically caused by misspellings, or missing or misplaced elements.
- Run-time errors, which arise from syntactically valid commands that for one reason or another cannot be carried out.

You can use the Debugger to identify and correct both of these kinds of errors. In addition, if the problem is a run-time error, you may be able to continue script play by interactively altering the sequence of commands or the context in which the script is running. If the problem is a syntax error, you will have to start the script over again after correcting the problem in the code.

2. Use the Editor to insert a DEBUG command in a script. When you play the script and the command is executed, you'll enter the Debugger. The current command will be the DEBUG statement.
3. Press *Ctrl-Break* to interrupt script play. Then choose Debug from the Script Break menu that appears. The current command will be the command after the one that was being executed when play was interrupted.

Pressing *Ctrl-Break* causes PAL to display the Script Break menu at the next statement boundary. For example, if you press *Ctrl-Break* to interrupt script play during a WAIT, you will hear a beep. If you then press one of the keys on the UNTIL list or issue an event on the *EventList* of the event-driven WAIT, the Script Break menu will appear.

4. Press PAL Menu *Alt-F10* and choose Debug. You'll be asked which script you want to debug. The current command will be the first one in the script.

The first three methods let you debug the current script (the one being played). You can use the fourth to play and debug any script.

---

## Working with the Debugger

The Debugger is unobtrusive, using only the bottom two lines of the screen to keep track of the commands in the script you're testing. Script play itself is completely under your control.

Apart from executing the instructions in the script you are debugging, you can control Paradox and PAL through either the regular Paradox menu (press Menu *F10*) or the Debugger menu (press PAL Menu *Alt-F10*).

---

### Debugging levels

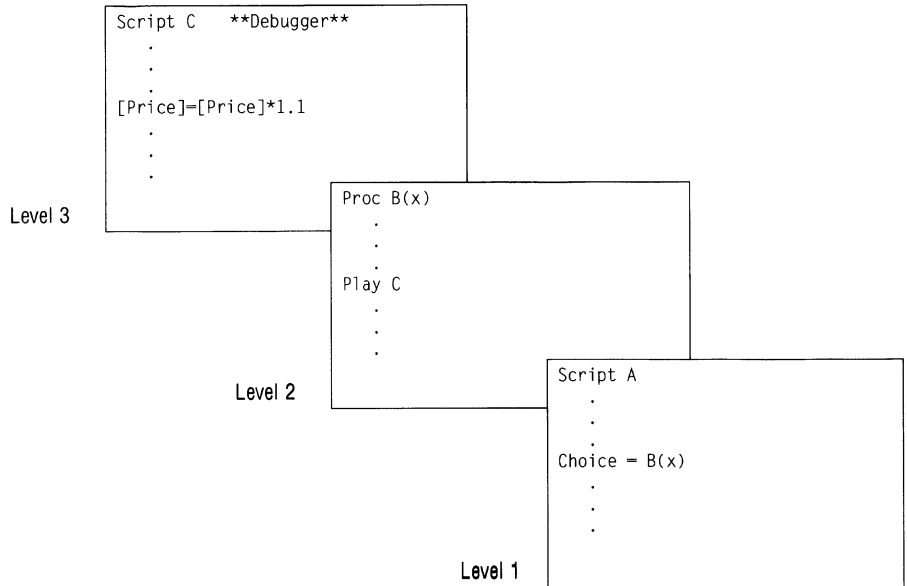
With the PAL Debugger, you can debug scripts and procedures nested to many levels. Suppose your application began with script *A* (level 1), which then called procedure *B* (level 2), which in turn called script *C* (level 3), which triggered a script error. As shown in Figure 10-1, each time a script or procedure is first played or called, the debugging level is increased by 1. Each time a script or procedure finishes execution, the debugging level is reduced by one. The Value and MiniScript commands on the Debugger menu (described later in this chapter) both increase the debugging level by 1 while they are being executed.

Seen as a whole, all levels of script or procedure control are called the *call chain*. When script *A* calls procedure *B*, script *A* is higher on the call chain than procedure *B*. Likewise, procedure *B* is lower on the call chain than script *A*.

A debugging session lasts until the script or procedure at debugging level 1 is played out, or until you exit with one of the methods discussed in "Quitting the Debugger" later in this chapter. It's often important to keep track of which level you are currently debugging. You can use the Debugger's Where? feature to see exactly what debugging level you are currently working on, and the Pop feature to leave the current level and return to the calling script. Both of these features are described later in this chapter.

Remember that some variables can be private to a procedure while others are global to all scripts and procedures. A change you make to a variable using a miniscript may or may not apply to other debugging levels, depending on where you are.

Figure 10-1 Debugging levels



---

## Debugging procedures from libraries

If you're playing a procedure from a library, you can go into Debug mode even if the source code for the procedure is not available; however, you will get a message saying "Source unavailable" and you will be unable to change the source. See Chapter 6 for more information about procedure libraries.

---

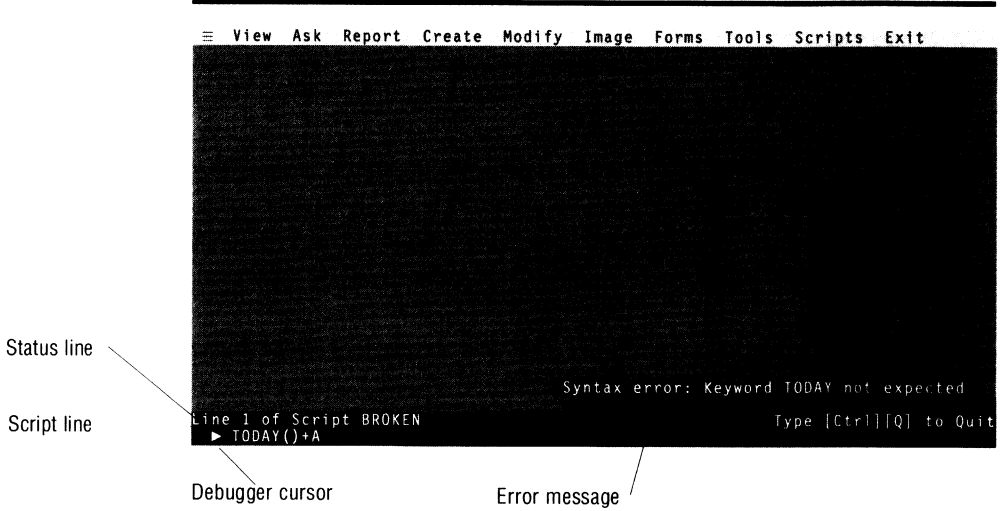
## The Debugger screen

You'll know when the Debugger is active because it presents status information about script execution on the bottom two lines of your screen:

- ▮ The Debugger *status line* identifies the script being executed and the current line number. The current script may not be the one you started with, since that one may have called others.
- ▮ The Debugger *script line* shows the current line of the script being executed. Since this can contain several commands, the Debugger cursor—an arrow—points to the next command to be executed.

Since the bottom lines are occupied, error messages are displayed in a window two lines higher than the status line. Otherwise, the screen looks just the same as when the Debugger is not active.

Figure 10-2 The Debugger screen



## Debugging keys

Table 10-1 shows the keys you can use to execute debugging commands while the Debugger is active. Except for PAL Menu *Alt-F10*, which displays the Debugger menu, all the keys duplicate commands on the Debugger menu. See the discussion of those commands later in this chapter for more information.

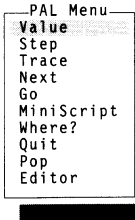
You can use these keys only when the Debugger is active.

Table 10-1 Debugging keys

Keypress	Name	Action
Ctrl-E	Editor	Edits script
Ctrl-G	Go	Resumes script play
Ctrl-N	Next	Skips current command and moves to the next
Alt-F10	PAL Menu	Displays Debugger menu
Ctrl-P	Pop	Moves up a level to the calling script
Ctrl-Q	Quit	Exits Debugger
Ctrl-S	Step	Executes current command
Ctrl-T	Trace	Traces through procedure
Ctrl-W	Where?	Shows current nesting levels

---

## The Debugger menu



While the Debugger is active, you can press PAL Menu *Alt-F10* any time to display the Debugger menu. This menu contains ten commands:

- Value: Calculates the value of an expression or variable.
- Step: Executes the current script command, but runs procedure calls, the PLAY command, the EXECUTE command, and the EXECPROC command as a single step no matter how many commands they contain.
- Trace: Executes the current script command and traces into procedures.
- Next: Skips the current command and moves to the next command.
- Go: Resumes continuous script play.
- MiniScript: Composes and executes a short PAL script.
- Where?: Shows current script and procedure nesting (the call chain).
- Quit: Leaves the Debugger.
- Pop: Stops playing the current script or procedure and moves up to the calling script.
- Editor: Edits the current script with the Editor.

You can't use Trace, Step, Next, or Go if you entered the Debugger as a result of a syntax error. All other functions are available no matter how you entered the Debugger.

---

### Value

Value—like its namesake on the PAL menu (see Chapter 8)—is a powerful calculator with which you can determine the value of any PAL expression. When you choose it from the Debugger menu, a dialog box asks you to type the expression you want to evaluate.

You can type up to 175 characters, and use the arrow keys with or without the *Ctrl* key to scroll through and edit the expression. (See the *User's Guide* for details about using the arrow keys when editing.)

You can redisplay and edit the string most recently typed for the Value or Miniscript command. After you've used either command at least once, choose it again, then press *Ctrl-E* at the prompt to redisplay the string. Also, you can append the most recently typed string to an existing string.

When you press *Enter*, PAL calculates and displays the value of the expression in the message window at the bottom right corner of the screen. The contents of the message window remain until you press another key.

Value is useful for determining whether a variable, array element, or expression is causing a problem in a script. If the current debugging level is a procedure, you can interrogate the values of its formal parameters and private variables as well.

Here are some examples of expressions you might interrogate in a debugging session:

- *x*, to see the value currently assigned to variable *x*
- NKEYFIELDS(TABLE()), to show the number of key fields in the current table
- TIME(), to know when to break for lunch

Like Value from the PAL menu, Debug | Value runs a short script called *Value*, which contains the command

```
RETURN Expression
```

where *Expression* is the PAL expression you asked to be evaluated. Thus using Value can itself cause a script error to occur; for example, if the expression contains a syntax error or an unassigned variable. In that case, you will find yourself in a debugging session within a debugging session. The status line will show that *Value* is now the script being debugged, and the script line will show the erroneous expression.

At that point, you can

- Press Pop *Ctrl-P* to pop back up to the script you were debugging (see the following discussion of Pop).
- Use MiniScript, if appropriate, to correct the error condition (for example, by assigning a variable).
- Press *Ctrl-E* to edit the most recent string.

---

## Step

Choosing Step or pressing *Ctrl-S* causes PAL to execute the current script command and advances the Debugger cursor to the next command on the script line. The only difference between Step and Trace is that Step steps over procedures and calls to PLAY, EXECUTE, and EXECPROC. The procedures or scripts are executed, but not shown line by line. Step is handy when you've thoroughly debugged your procedures and don't want to trace into them every time they're called, or when you want to get an overview of your application by moving through it quickly.

---

## Trace

Choosing Trace or pressing Trace *Ctrl-T* causes PAL to execute the current script command and advances the Debugger cursor to the next command on the script line. You can use Trace to play the current script or procedure, one command at a time.

What you see when you trace through a script depends on the command:

- ❑ Commands that make changes to the workspace or perform input or output will have visible effects on your screen with each trace.
- ❑ Other commands, such as assignment to variables, will produce no other visible effect than to advance the Debugger cursor. (You can use Value to check the current value of a variable after an assignment statement.)
- ❑ If you trace through a procedure definition, all you will see is the procedure header (the PROC command). Later, when your script actually calls the procedure, you will be able to trace through each of the commands in its body.
- ❑ If you trace through an EXECUTE command, each command in the EXECUTE script will be processed separately.

If you entered the Debugger because a run-time error occurred, the command executed when you choose Trace is the same one that produced the error. Unless you first correct the condition that caused the error, it will most likely recur, and Trace will have no visible effect. You can also choose Next from the Debugger menu or press Next *Ctrl-N* to bypass a command causing a run-time error.

If you entered the Debugger because of a syntax error, you can't use Trace at all. You must correct the problem using the Editor and then replay the script from the beginning.

---

## Next

Choosing Next or pressing Next *Ctrl-N* skips the current command and advances the Debugger cursor to the next one. It is most useful when you want to bypass a command that's causing a run-time error.

You may want to run a miniscript to correct the error situation before using Step or Go to resume the script. Remember to use the Editor to correct the erroneous command before the script is played again.

As with Trace and Step, you can't use Next to bypass a command that causes a syntax error.

---

## Go

Choosing Go or pressing Go *Ctrl-G* resumes continuous script play starting with the current command. If you are in a nested debugging context when you choose Go, script play continues until the end of the top-level script, the next error, or the next Debug command.



As with Step, Trace, and Next, you can't use Go if you entered the Debugger on a syntax error. If you entered on a run-time error, make sure to correct the cause of the error before choosing Go, or the error will be retrigged immediately.

---

## MiniScript

MiniScript lets you compose and execute a small script (up to 175 characters). This command works very much like MiniScript on the PAL menu (see Chapter 8). When you choose MiniScript from the Debugger menu, a dialog box asks you to type the script. You can enter any valid script commands, except procedure definitions, up to a limit of 175 characters, and use the arrow keys with or without the *Ctrl* key to scroll through and edit the miniscript. (See the *User's Guide* for details about editing.)

You can redisplay and edit the string most recently typed for the Value or Miniscript command. After you've used either command at least once, choose it again, then press *Ctrl-E* at the prompt to redisplay the string. Also, you can append the most recently typed string to an existing string.

The main difference between MiniScript and Debug | MiniScript is that Debug | MiniScript does not execute your miniscript immediately when you press *Enter*. Instead, the miniscript becomes the current script—nested one level down—and is displayed in the Debugger window at the bottom of the screen. To play the miniscript, use either Step (single step), Trace, or Go (continuous execution). When the miniscript has finished, the debugging level returns to the next higher script—the one you were debugging when you chose MiniScript. If you change your mind and want to abandon the miniscript, use Pop *Ctrl-P*.

Miniscripts are especially useful for temporary fixes to recover from a run-time error. For example, if you neglected to assign a variable before using it in an expression, a run-time error occurs when the expression is evaluated. You can enter the Debugger, use a miniscript to assign a value to the variable, and then use Go to resume script play. Remember to use the Editor to correct the script later.

The commands you type in a miniscript are actually executed as a short script called *Mini*. As with Value, if a script error occurs during the course of the play of a miniscript, you will reenter the Debugger one level below the original debugging level. You can use Pop *Ctrl-P* to pop back to the previous context at this point. You can't use the Editor to edit a miniscript.

If a variable has a private value in the debugging context from which you choose MiniScript, any reference to it in a miniscript refers to the private variable.

---

## Where?

Choosing Where? or pressing Where? *Ctrl-W* shows you a picture of the current nesting of scripts and procedures, sometimes referred to as the *call chain*. It also shows the values of all formal parameters and private variables defined in procedures, and the values imported into a closed procedure with the USEVARS option. Press any key to resume debugging.

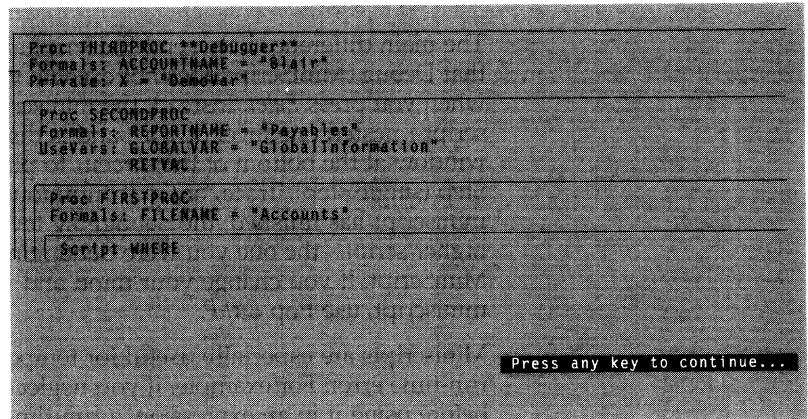
Where? is useful in nested applications for tracking down which script or procedure contains the error you're looking for. It's also handy when you've triggered an error within an error and aren't quite sure of the current debugging level.

If your scripts are nested many levels, the Where? display may not fit on a single screen. If so, pressing any key displays the next screen. You must display all the Where? screens in this way before resuming or exiting from the debugging session.

Figure 10-3 Finding out where you are

---

Procedure parameters and variables (current level)



```
Proc: THIRDPROC - **Debugger**
Formals: ACCOUNTNAME = "01air"
Private: X = "Demovar"

Proc: SECONDPROC
Formals: REPORTNAME = "Payables"
UseVars: GLOBALVAR = "GlobalInformation"
         RETVAL

Proc: FIRSTPROC
Formals: FILENAME = "Accounts"

Script: WHERE

Press any key to continue...
```

---

## Quit

Choosing Quit or pressing Quit *Ctrl-Q* exits the Debugger and cancels all levels of script play.

Choosing this command releases all procedure definitions, as when any top-level script play is terminated. Values assigned to global variables and arrays are retained, however.

---

## Pop

Choosing Pop or pressing Pop *Ctrl-P* abandons play of the current script and pops you up to the script that called it (if any). If the current script is the top-level script, Pop has the same effect as Quit.

Pop is especially useful for popping out of an error that occurred in a Value or MiniScript, since both of these commands work by playing a short script.

---

## Editor

Choosing Editor or pressing *Ctrl-E* exits from the Debugger and takes you directly to the Editor. The script you were debugging appears, and the cursor is placed at the first instruction on the line that was current. The Editor is described in the *User's Guide*.

If you've used the Custom Configuration Program to tell Paradox that you want to use another editor, your editor will be invoked instead of the Paradox Editor.

When you're finished editing the script, choose Go from the Editor menu to replay it. This option is, of course, not available from your own editor. You can emulate it, however, by pressing Go *Ctrl-G* immediately after returning from your editor to Paradox. If you want to debug the script, choose Debug from the PAL menu to run the script, or put a DEBUG command at the point where you want debugging to start.

---

## Quitting the Debugger

There are several ways of ending a debugging session and exiting the Debugger. Two menu commands (and their keyboard equivalents) always end Debug mode:

- Quit or Quit *Ctrl-Q* exits the Debugger.
- Editor *Ctrl-E* exits the Debugger and enters the Editor at the current line (if you are running a script) or returns to a MiniScript type-in box.

Several other menu commands and key combinations might terminate Debug mode, depending on how and when they're used:

- Four commands exit the Debugger if they come to the end of the top-level script:
  - Trace or Trace *Ctrl-T*
  - Step or Step *Ctrl-S*
  - Next or Next *Ctrl-N*
  - Go or Go *Ctrl-G*
- Pop (*Ctrl-P*) exits the Debugger when you use it in the top level of script play.

In addition, several PAL commands will end a debugging session if they are executed in a script you're debugging:

- ❑ RETURN, if used in a top-level script
- ❑ QUIT
- ❑ EXIT (this command also ends the Paradox session)

Leaving the Debugger has the same effect as completing play of a top-level script:

- ❑ The Paradox workspace reappears.
- ❑ All procedure definitions are released.
- ❑ All variables and arrays retain their values.

---

## A Debugger tutorial

Let's use a sample script to demonstrate some of the features of the Debugger and illustrate tips on how to use it.

If you have not copied the sample tables to your hard disk or to your Paradox private directory, follow these instructions:

- ❑ For standalone computers, place the Paradox Installation/Sample Tables Disk in drive A. Use the installation program's Optional Software Installation option to re-install the sample tables.
- ❑ If your system is connected to a network, use the Paradox Tools | Copy command to copy the *Debugtst* script from the Paradox shared data directory to your Paradox private directory. If *Debugtst* is not there, ask your network administrator to install it.

**Note** If you've used CCP to link another editor into Paradox instead of the PAL Editor, you will need to adapt the editing instructions in the tutorial to work with the editor you're using.

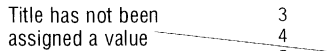
### Example 10-1 A test script

---

Here's a listing of the *Debugtst* script:

```
1 ; DEBUGTST--Note: This script has errors in it!
2 PROC SomeText() ; define a procedure that displays text
3 @4,0 ; locate cursor
4 STYLE REVERSE ; change to reverse video
5 ? title ; print title of the screen
6 STYLE ; return to normal text display
7 @7,0 ; relocated cursor
8 TEXT
9
10 This is just some miscellaneous text to occupy the
11 screen during our debugging test. It should appear
approximately centered, a couple of lines below the
```

Title has not been assigned a value



```

12             title for the screen.
13     ENDTXT
14     SLEEP 5000           ; wait 5 seconds
15 ENDPROC
16
17 WHILE(TRUE)           ; just a test loop, always true
18     CLEAR               ; clear the PAL canvas (screen)
19     CLEARALL            ; make sure no images are left on workspace
20     @4,0                ; locate cursor at line 4, column 0
21     SHOWPOPOP "School Days" CENTERED ; build a fake user menu
22     "Test" : "This is just a test menu item":"Test",
23     "Quiz" : "Yet another test menu item":"Quiz",
24     "Exam" : "Another test menu item":"Exam"
25     "Exit" : "Leave the test application, thank goodness":"Exit"
26 ENDMENU TO choice
27 IF choice = "Esc"
28     THEN QUIT
29 ENDIF
30 CLEAR
31 SWITCH
32     CASE choice = "Test":
33         SomeText()
34     CASE choice = "Quiz":
35         SomeText()
36     CASE choice = "Exam":
37         SomeText()
38     CASE choice = "Exit":
39         QUIT
40     ENDSWITCH
41 ENDWHILE

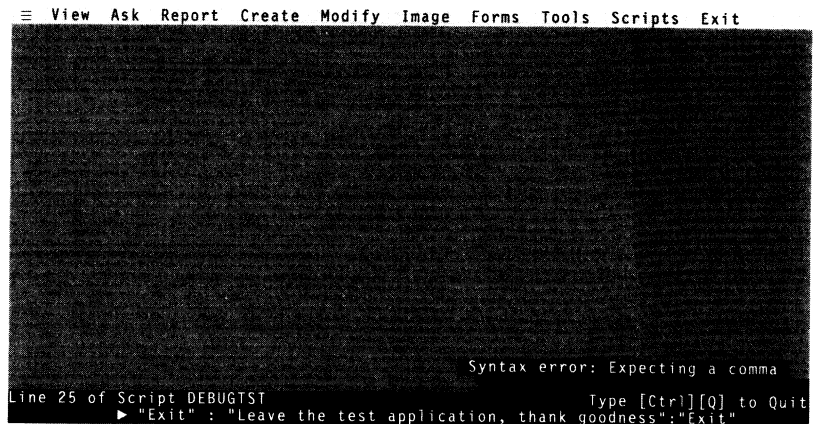
```

Missing comma at  
end of line

### Example 10-2 Debugging a script

1. Press PAL Menu *Alt-F10* and choose Debug from the PAL menu. Type *Debugtst* as the script you want to debug.

The *Debugtst* script should begin by clearing the canvas and workspace, then starting a WHILE loop. But the SHOWPOPOP command, the first substantive thing the script has to do, has a syntax error.



Your screen should look like the one shown here. You see the Debugger status line and script line at the bottom of the screen.

Line 25 of the script has an error. Notice the error message Expecting a comma. If you examine the *Debugtst* script in Example 10-1, you'll see that we've omitted a comma at the end of the previous line (line 24), with the Exam menu choice.

**Note** When looking for an error, take a close look at the end of the previous line, as well as the line where the error occurs.

2. To call the Editor to fix the error, press Editor **Ctrl-E**. When the script is displayed, notice that the cursor is on the line where the Debugger detected the error.

Insert a comma at the end of the previous line so the SHOWPOPUP command looks like the example shown here.

```
SHOWPOPUP "School Days" CENTERED ; build a fake user menu
  "Test" : "This is just a test menu item":"Test",
  "Quiz" : "Yet another test menu item":"Quiz",
  "Exam" : "Another test menu item":"Exam",
  "Exit" : "Leave the test application, thank goodness":"Exit"
ENDMENU TO choice
```

Insert  
missing  
comma

3. Press Do-It! **F2** to finish editing the script, save it on disk, and return to the Paradox workspace.

You could also have chosen Go from the Editor menu, which saves the modified script and then plays it, but you want to go back into Debug mode.

4. Press PAL Menu **Alt-F10**, choose Debug, and choose the *Debugtst* script again.

5. This time, trace through the script by pressing Trace **Ctrl-T**.

As you continue to press Trace **Ctrl-T**, notice that the Debugger status line and script line are updated after each step. Sometimes when you press Trace **Ctrl-T**, you'll see some action (as in line 18, when CLEAR executes and clears the current PAL canvas). Sometimes nothing will happen (as in line 2, when PROC *SomeText()* is defined).

Notice as you trace that some lines, like the body of the *SomeText* procedure, seem to be skipped.

Actually they weren't—procedures execute when they are called, not when they are defined. Paradox noted the PROC statement, added *SomeText* to the active procedure list, and jumped to the first command (WHILE) after the procedure definition.

When you reach the SHOWPOPUP command, a menu appears. Pressing Trace **Ctrl-T** again results only in a beep. This is because PAL is waiting for input; in this case, you must make a menu choice before it can continue.

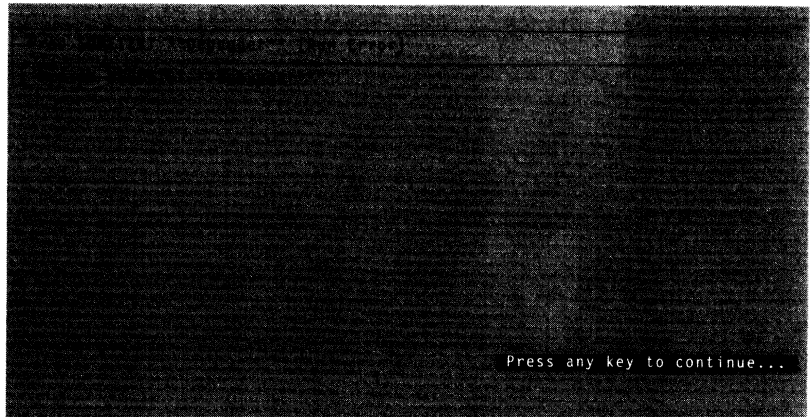
Again, the separate lines of the SHOWPOPUP command don't appear on the script line (unless there is an error) because they are all part of the same SHOWPOPUP construct.

6. Choose Quiz from the menu and continue to Trace *Ctrl-T* through the script.

Since the script calls *SomeText*, you'll begin to see the lines of this procedure. Then you'll hit another error, at the command *? title*.

7. To find out where you are, press Where? *Ctrl-W*. You'll see the Where? display.

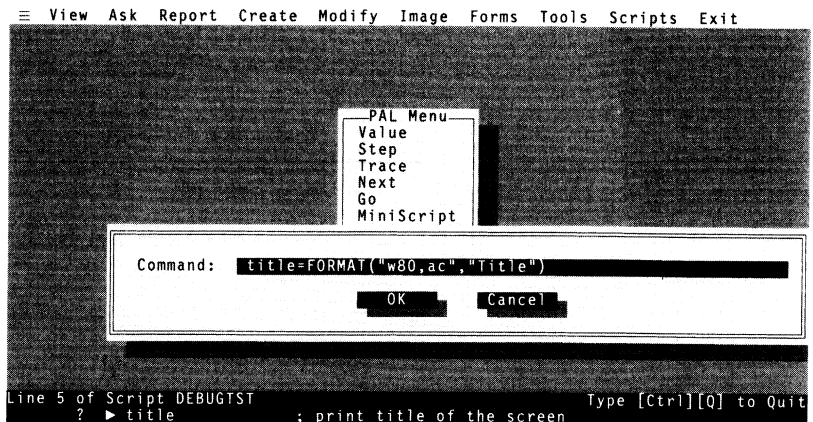
Notice that you're a couple of levels down. You started by playing the *Debugtst* script (the bottom level). *Debugtst* contains the procedure *SomeText* (level 2), which was being executed when the error occurred.



8. Press any key to return from the Where? display. At the bottom of the screen, you'll see that the error is on line 5 of *Debugtst*.

The Debugger cursor indicates the *? title* command and the variable *title*. The error is obvious: We forgot to initialize *title* by assigning a value to it.

9. To give *title* a temporary value so that you can continue to test the script, press PAL Menu *Alt-F10* and choose MiniScript from the Debugger menu. Type in the missing assignment statement as shown here and press *Enter*.



10. To find out where you are again, press Where? *Ctrl-W*. You should see the Where? display again.

Recall from the discussion of MiniScript that a miniscript is executed as a script. That's why "Script Mini" appears as a level in the Where? display.

11. Press any key to resume the Debugger from the Where? display.
12. Since the miniscript is the current script line, press Trace *Ctrl-T* to execute it and give *title* a value.

Sure enough, repeatedly pressing Trace *Ctrl-T* gets you past the original error, displays the correct *title*, and then displays the text from *SomeText*:

```
Title
this is just some miscellaneous text to occupy the
screen during our debugging test. It should appear
approximately centered, a couple of lines below the
title for the screen.

Line 14 of Script DEBUGST ; wait 5 seconds
Type [Ctrl][Q] to Quit
```

13. Press Trace *Ctrl-T* again when the SLEEP 5000 command appears. Notice that nothing seems to happen for a few seconds. That's because Paradox is executing the SLEEP command, which causes the system to pause for a designated amount of time—in this case, five seconds.
14. Continue to press Trace *Ctrl-T* until you see the Test menu again. This time choose Exit and continue tracing. Shortly you'll see the QUIT command appear on the Debug Script Line. This is a separate line in the SWITCH structure that is executed in case Exit is chosen.
15. Press Trace *Ctrl-T* to step through the QUIT command. This ends all script play and returns you to the Paradox Main menu and workspace. You are now done testing *Debugst*.
16. But you're not quite done correcting the script! Remember that you had to use a miniscript to assign a value to *title* in order to get the script to run correctly. Since doing so seems to have corrected all the remaining errors, press Editor *Ctrl-E* to start the Editor, and add the missing line:

```
title = FORMAT("w80,ac","Title")
```



You must insert this line at some point before the ? title command; exactly where, we'll leave up to you.

**Caution** Remember that, once defined, variables stay defined for the duration of a Paradox session. If you tried playing the script at this point without correcting the error, it would play correctly despite the fact that *title* is still not assigned within the script. That's because Paradox still remembers *title*'s value from the debugging session. In general, before testing one script after playing another, execute the miniscript

```
RELEASE VARS ALL
```

which assures that no variables have values left over from previous script play.



# Interacting with the user

PAL lets you create and control a user interface that includes pull-down menus, pop-up menus, and dialog boxes. The event-driven programming model of Paradox lets users interact with these interface elements.

Part III consists of seven chapters:

- ❑ Chapter 11, “Using windows.” This chapter describes how to create windows, control their appearance, and manage them.
- ❑ Chapter 12, “Controlling screen display.” This chapter discusses PAL canvases and the echo layer.
- ❑ Chapter 13, “Handling events.” This chapter describes the event-driven programming that is central to Paradox.
- ❑ Chapter 14, “Creating menus.” This chapter describes the different types of PAL menus you can create to interact with the user.
- ❑ Chapter 15, “Creating dialog boxes.” This chapter describes using the `SHOWDIALOG` command to create dialog boxes and using other commands to control dialog boxes.
- ❑ Chapter 16, “Using the `WAIT` command.” This chapter describes using the `WAIT` command to pause processing while the user interacts with a field, record, table, form, or the entire workspace.
- ❑ Chapter 17, “Using multi-table forms.” This chapter explains how to use Paradox’s sophisticated multi-table forms in your applications. It also discusses Paradox’s built-in referential integrity checking.



# Using windows

PAL lets you create and open windows for canvases and Paradox objects. By understanding how Paradox is divided into layers for different types of objects, you can control and manipulate these windows.

This chapter covers

- working with windows
- using the three layers of Paradox
- creating floating windows

---

## Where the action takes place

In PAL applications, just as with interactive Paradox, most of the action takes place in windows. Menus and dialog boxes also play important roles, but they are used to solicit user input and are discussed in Chapter 14 and Chapter 15. Table A-18 in Appendix A of the *PAL Reference* summarizes the window management commands and functions available in Paradox. The following sections provide a detailed explanation of what windows are, what they contain, and how to use them.

---

## Objects in windows

Windows can contain different kinds of objects, including tables in table or form view, report or form designs, sort forms, PAL canvases, and so on. The type of object that a window holds does not affect the essential behavior of the window. A PAL canvas, like other Paradox objects, is placed in a window. The PAL canvas is not a special kind of window; it is a special kind of object in a regular window.

---

## Creating a window

You create PAL canvas windows explicitly with the `WINDOW CREATE` command. Image windows are created automatically when you view an object. You can create a PAL canvas window with the `WINDOW CREATE` command as follows:

```
WINDOW CREATE TO DisplayWindow
```

Whenever Paradox creates or opens any type of window, the window is assigned a unique integer value called the *window handle*. The `WINDOW CREATE` command places the value of the window handle in a variable that you specify. In the example above, *DisplayWindow* is the variable that contains the window handle. The value of the variable evaluates to the window handle, so the variable is a convenient way to refer to the window. Window handles are discussed in more detail later in this chapter.

If you want, you can specify the location of the upper left corner of the window when you create it. The statement

```
WINDOW CREATE @1,0 TO DisplayWindow
```

creates the canvas window and places the upper left corner at row 1, column 0 of your screen.

You can optionally specify the height and width of a canvas window at the time of creation. The statement

```
WINDOW CREATE HEIGHT 23 WIDTH 80 @1,0 TO DisplayWindow
```

creates a window that is 23 lines high and 80 columns wide. Remember that this is not necessarily a full-screen window on all monitors.

The `WINDOW CREATE` command also uses the optional keywords `ATTRIBUTES` and `FLOATING`. The `ATTRIBUTES` keyword lets you specify all the attributes of a window from a dynamic array; the `FLOATING` keyword lets you create a window above all other objects. See “Working with window attributes” and “Floating windows” later in this chapter for a description of the `ATTRIBUTES` and `FLOATING` keywords.

---

## Getting a window handle

When Paradox opens any window, the window is assigned a unique number called the window handle. Before you can set the attributes of a window, move to it, or close it—in fact, before you can do much of anything to a window—you need to know the window handle.

Handles are the best way to keep track of windows. Every time you open a window, you should get its handle immediately. Paradox encourages users to move freely from window to window, relocating and resizing them at will. You should always get a handle for a window as it is opened because you cannot tell what a user will do after a window becomes available.

If you create a window explicitly with the WINDOW CREATE command, the window handle is assigned to the variable you specify in the command. For example,

```
WINDOW CREATE @1,0 WIDTH 80 HEIGHT 10 TO SplashScreen
```

creates a canvas window and puts its handle in the variable *SplashScreen*.

If you open a table on the desktop with a command such as Menu {View} {Customer}, the table is placed in a window, but you need to use the WINDOW HANDLE command to get its handle as follows:

```
WINDOW HANDLE IMAGE 1 TO CustomerTable
```

If you know that the table window is the current window, you can use the CURRENT keyword instead of IMAGE 1:

```
WINDOW HANDLE CURRENT TO CustomerTable
```

You can also use the GETWINDOW() function to get a handle for the current window:

```
CustomerTable = GETWINDOW()
```

In all these cases, the handle for the *Customer* table is assigned to the variable *CustomerTable*. The GETWINDOW() function gets the same handle as WINDOW HANDLE CURRENT does. Because GETWINDOW() is a function, however, it can be used in expressions.

Use WINDOW HANDLE with the DIALOG keyword inside a dialog procedure to get a handle for a SHOWDIALOG dialog box:

```
WINDOW HANDLE DIALOG TO DialogBox
```

Because a SHOWDIALOG dialog box is not current when the dialog procedure is executing, neither WINDOW HANDLE CURRENT nor GETWINDOW() will return the handle of a dialog box.

To get a handle for every window that is open, use the WINDOW LIST command to create a fixed array:

```
WINDOW LIST TO WindowHandles
```

The *WindowHandles* array contains an index for each open window; the value of each index in this array is the window handle.

See the *PAL Reference* for a complete description of the syntax and options of WINDOW HANDLE and WINDOW LIST. See Chapter 15

for information about getting a handle for a dialog box created with the PAL SHOWDIALOG command.

---

## Testing a handle

A window handle is used only once in a Paradox session. If a window is closed, its handle is not used again.

Because users can close a window, your application could attempt an operation with an invalid handle and cause a script error. To make sure that the window handle you are using is assigned to an active window, you can use the ISWINDOW() function. The following example opens the *Bookord* table, assigns the variable *BookordWindow* to the window handle, tests that handle, and displays a message about the window.

```
VIEW "Bookord"  
BookordWindow = GETWINDOW()  
IF ISWINDOW(BookordWindow)  
    THEN MESSAGE "Window ", BookordWindow, " is alive."  
    ELSE MESSAGE "Window ", BookordWindow, " is gone."  
ENDIF
```

---

## Working with window attributes

When Paradox creates a window, that window has a set of attributes that tell Paradox everything it needs to know about the window. These attributes can be stored in a dynamic array. You can examine the elements of that array to discover what you need to know about such things as the window's location (origin), size, title, and style. You can also examine the settings that determine whether a window is allowed to be moved or closed.

Likewise, you can tell Paradox to set the attributes of a window from the contents of the dynamic array. This gives you tremendous control over windows with a minimum of programming effort.

---

## Getting window attributes

In this example, we'll use the WINDOW GETATTRIBUTES command to get a window's attributes and store them to a dynamic array. Then we'll use a FOREACH loop to display each element in the array.

```
VIEW "Products"  
    ; get a handle for the Products table  
WINDOW HANDLE CURRENT TO ProductWindow  
    ; create a dynamic array for ProductWindow window attributes  
WINDOW GETATTRIBUTES ProductWindow TO ProductAttributes  
ECHO NORMAL  
    ; create a canvas window to display ProductWindow attributes  
WINDOW CREATE @1,0 HEIGHT 23 WIDTH 25 TO DisplayCanvas  
FOREACH Element IN ProductAttributes  
    ? Element, " = ", ProductAttributes[Element]  
    SLEEP 1000  
ENDFOREACH
```



If you run this script, you'll see output similar to the following:

```
CANCLOSE = True
CANMAXIMIZE = True
CANMOVE = True
CANRESIZE = True
CANVAS = True
CANVASHEIGHT = 16
CANVASWIDTH = 71
ECHO = True
FLOATING = False
HASFRAME = True
HASSHADOW = True
HEIGHT = 18
MARGIN = "OFF"
MAXIMIZED = False
ORIGINCOL = 1
ORIGINROW = 2
SCROLLCOL = 0
SCROLLROW = 0
STYLE = 112
TITLE = "Products"
WIDTH = 73
```

Because the window attributes are stored in a dynamic array, the sequence in which they are displayed is not necessarily the same as the order in which they are listed here. In this display, the item on the left is the dynamic array index (also called a tag) for the window attribute; the item on the right is the value of the index. Each index in the array controls a specific attribute of the window. Table 11-1 below summarizes these window attributes.

Table 11-1 Window attributes

Attribute	Description
CANCLOSE	True if user is allowed to close; False otherwise
CANMAXIMIZE	True if user is allowed to maximize and restore; False otherwise
CANMOVE	True if user is allowed to move; False otherwise
CANRESIZE	True if user is allowed to resize; False otherwise
CANVAS	True if canvas writing commands will be displayed immediately on the canvas; False if the display will be delayed
CANVASHEIGHT	Height of the canvas buffer
CANVASWIDTH	Width of the canvas buffer
ECHO	True if WINDOW ECHO is True; False otherwise
FLOATING	True if a canvas window is above the echo layer; False otherwise
HASFRAME	True if window is framed; False otherwise
HASSHADOW	True if a shadow exists; False otherwise
HEIGHT	Number of rows in the height including the frame

Attribute	Description
MARGIN	Number representing left margin setting for display of text on the canvas; "OFF" if none
MAXIMIZED	True if maximized; False otherwise
ORIGINCOL	Column number of the upper left corner relative to the screen
ORIGINROW	Row number of the upper left corner relative to the screen
SCROLLCOL	Offset of the horizontal scroll bar to the right
SCROLLROW	Offset of the vertical scroll bar down
STYLE	Style attribute number for canvas text
TITLE	Name that appears centered at the top of the frame
WIDTH	Number of columns in the width including the frame

## Setting window attributes

You can also set most of the attributes of a window from a dynamic array. You can specify all the window attributes in this array, but it is more efficient to specify only the ones you want to change.

Here's how you would change the title of the window *ProductsWindow*:

```
DYNARRAY WindowAttributes[] ; create a new dynamic array
    ; specify an index and value for the title attribute
WindowAttributes["TITLE"]="Stock"
    ; set the attributes of the ProductsWindow window
WINDOW SETATTRIBUTES ProductsWindow FROM WindowAttributes
```

In an earlier section of this chapter, we discussed creating a window with the following statement:

```
WINDOW CREATE @1,0 WIDTH 80 HEIGHT 10 TO SplashScreen
```

If you specify window attributes in a dynamic array before you create the window, you can use the optional ATTRIBUTES keyword to set the attributes of a window when you create it, as follows:

```
DYNARRAY StandardAttributes[] ; create a dynamic array
StandardAttributes["ORIGINROW"] = 1 ; specify attributes
StandardAttributes["ORIGINCOL"] = 0
StandardAttributes["WIDTH"] = 80
StandardAttributes["HEIGHT"] = 10

; create the window and set the attributes
WINDOW CREATE ATTRIBUTES StandardAttributes TO SplashScreen
```

WINDOW SETATTRIBUTES is frequently used to make one window look like another window or to establish preferences for windows. WINDOW SETATTRIBUTES lets you set any or all of the attributes listed in Table 11-1 except SCROLLCOL, SCROLLROW, CANVASWIDTH, and CANVASHEIGHT.

If you use WINDOW GETATTRIBUTES to save default window attributes and then reset the attributes with WINDOW

SETATTRIBUTES, the window contents will not scroll to an obsolete location because the SCROLLCOL and SCROLLROW tags will be ignored. Use the WINDOW SCROLL command to specify the offset of a window. Using the SCROLLROW and SCROLLCOL tags with WINDOW SETATTRIBUTES will have no effect. See “Sizing and shaping windows” in this chapter for information about the WINDOW SCROLL command.

You cannot use the WINDOW SETATTRIBUTES command to specify a WIDTH and HEIGHT that exceed the CANVASWIDTH and CANVASHEIGHT of an unframed canvas window. CANVASWIDTH and CANVASHEIGHT cannot be used to change the dimensions of a canvas with the WINDOW SETATTRIBUTES command after a window has been created; however, you can use CANVASWIDTH and CANVASHEIGHT in a dynamic array and set the canvas dimensions when a window is created with the WINDOW CREATE command.

For more information about setting window attributes, see the WINDOW SETATTRIBUTES command in the *PAL Reference*.

---

## Sizing and shaping windows

In addition to setting window attributes from a dynamic array, you can control individual attributes with the commands WINDOW RESIZE, WINDOW MOVE, WINDOW MAXIMIZE, and WINDOW SCROLL. WINDOW RESIZE can work to either enlarge or shrink a window. WINDOW MOVE lets you change the location of a window. WINDOW MAXIMIZE is a toggle: It enlarges a window that is not maximized to the size of the desktop and restores a maximized window to its previous size. WINDOW SCROLL lets you specify the number of columns to the right and the number of columns down that the window has scrolled within the frame.

Following is an example of these commands:

```
VIEW "Backord"
WINDOW HANDLE CURRENT TO BackordWindow ; get a handle for the Backord window
ECHO NORMAL

WINDOW RESIZE BackordWindow TO 5,80 ; change size of window
SLEEP 1000

WINDOW MOVE BackordWindow TO 3,40 ; move the window on the screen
SLEEP 1000

WINDOW MAXIMIZE BackordWindow ; maximize the window
SLEEP 1000

WINDOW MAXIMIZE BackordWindow ; restore the window
SLEEP 1000

WINDOW SCROLL BackordWindow TO 5,5 ; scroll the window
```

## Specifying the current window

When you first open a window on the desktop, that window becomes the *current window*. The current window is the one that appears above other windows on the screen; it is the window that appears with the double-line border when echo is normal. A window must be *active* in order to be current. For example, if the menu is active, no window is current.

A user can change the current window by clicking with the mouse, using the `= | Window` menu command and selecting a window from the pick list, or choosing Next Window *Ctrl-F4*; your application can change the current window with PAL commands.

To make a window current, use the `WINDOW SELECT` command and supply it with a window handle, like so:

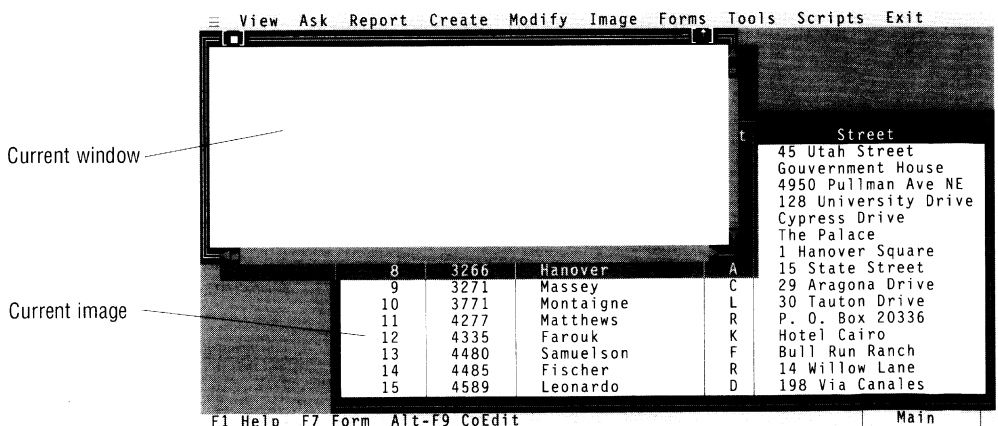
```
WINDOW SELECT ProductsWindow
```

The current window is not necessarily the same as the current image. If the current window is a canvas window, the current image is the Paradox image that was most recently the current window. You can have a current window and a current image that are different onscreen at the same time. For example,

```
VIEW "Customer"           ; the current image  
WINDOW CREATE TO DisplayWindow ; the current window
```

Figure 11-1 illustrates the difference between the current image and the current window.

Figure 11-1 Current image and current window



The distinction between current image and current window is important: in some situations, certain user interactions can affect the current image while other interactions affect the current window. Even though the current image may appear “deselected” (without the double-border) it is current as a Paradox object.

For example, you could have a WAIT table (the current image) available for a user to interact with while a canvas window (the current window) above it displayed help or messages. In this situation, the WAIT table would still be the current image, and keys pressed by the user would be passed to it and not to the current window. Remember that Paradox interactions with the current image will affect that image even if it is not the current window.

---

## Closing windows

You will periodically want to close the windows that you no longer need. To close the current window, you can use the WINDOW CLOSE command. To close all open windows except floating windows, issue the statement:

```
ALTSPACE {Desktop} {Empty}
```

If the system menu is not available or if you want to close floating windows also, you can use the following commands to select each window and close it:

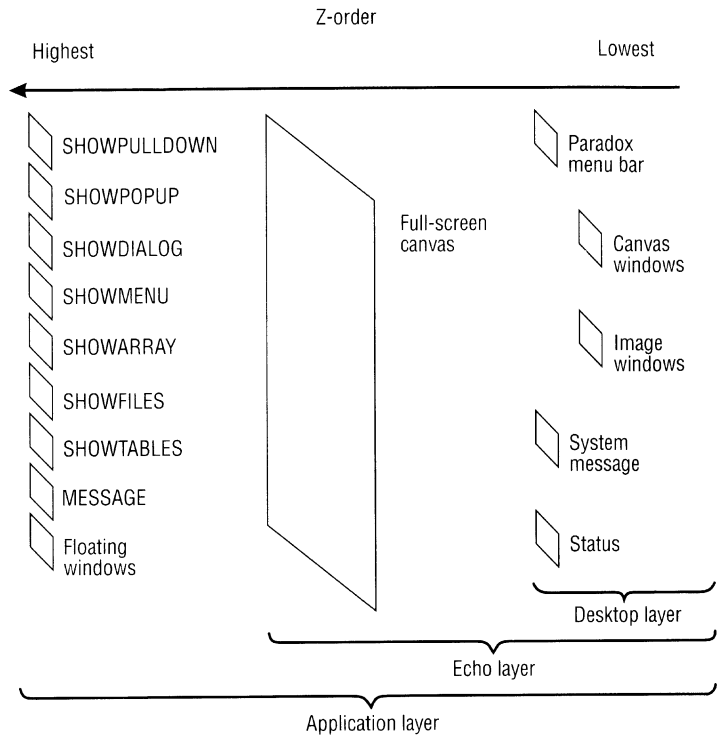
```
WINDOW LIST TO AllWindows  
FOR WindowHandle FROM 1 TO ARRAYSIZE(AllWindows)  
    WINDOW SELECT AllWindows[WindowHandle]  
    WINDOW CLOSE  
ENDFOR
```

---

## The three layers of Paradox

The *z-order* is the order in which objects onscreen appear to be stacked; it is the order of objects along the z-axis that runs from the screen out toward you. Figure 11-2 illustrates the z-order which Paradox maintains for different types of objects.

Figure 11-2 Paradox z-order

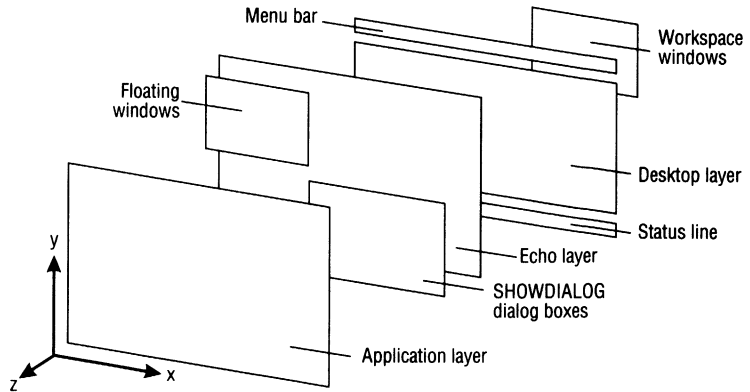


This z-order is divided into three layers:

- The desktop layer, where the Paradox menu and objects such as tables and forms reside. The desktop layer is the lowest layer.
- The echo layer, which controls the display of the desktop. The echo layer is above the desktop layer and contains it.
- The application layer, where interface objects that are created by an application reside. The application layer is the highest layer and contains both lower layers.

Figure 11-3 illustrates the type of objects in each layer and how each layer contains a lower layer.

Figure 11-3 Layers of the z-order



---

## The desktop layer

The Paradox desktop appears in the desktop layer as the shaded background underneath everything onscreen. The Paradox menu and the Paradox status line appear above the desktop in this layer.

Paradox objects always reside in the desktop layer. As canvas windows are created or opened, they also appear in the desktop layer unless they are created with the `FLOATING` keyword. Each new window is created or opened above any existing windows. By default, the z-order of windows is the order in which they are created or opened.

Users can change the z-order interactively by clicking windows with the mouse, using the `=|Window` command and selecting a window from the pick list, or using `Ctrl-F4`. When users or your application change the initial z-order, the z-order no longer reflects the order of creation.

Next `WindowCtrl-F4` makes the lowest window in the z-order active by raising it to the top of the z-order. `WINNEXT` is provided as a command equivalent for `Ctrl-F4`. Because z-order is relative, this command is not the best way for a PAL script to select a window. To maintain control in your application, you should always get a window handle when the window is created or opened and use the window handle with `WINDOW SELECT`.

---

## The echo layer

The echo layer is above the desktop layer and contains it. The echo layer controls the display of all objects in the desktop layer. The status of the echo layer is controlled with the `ECHO` command. When system `ECHO` is off, the display of the desktop is frozen; an image of

the screen at the point when ECHO is turned off is placed on the default full-screen canvas at the top of the echo layer. Changes that are made to anything under the echo layer, such as changes to images in the desktop, can still occur, but these changes are not displayed on the screen until ECHO is turned on again. When ECHO is turned on, changes that have occurred within the echo layer are visible. See Chapter 12, “Controlling screen display,” for a complete description of ECHO and the echo layer.

---

## The application layer

The application layer contains objects such as menus, status lines, and dialog boxes created by your application. Because the application layer is above the echo layer, objects in the application layer are not subject to system echo: They are always visible.

Objects in the application layer can hide objects in other layers. For example, your application can create a menu (even a menu with no items!) that will hide the Paradox Main menu.

---

## The workspace and the desktop

In Paradox, the workspace is where *objects*, such as tables, forms, and queries reside; the desktop is where *windows* reside. The workspace is an abstract concept. You do not have direct access to the workspace except through the desktop. A window is a viewing port that lets you look into a workspace object.

For example, if you issue the command `VIEW Customer`, the *Customer* table is placed on the workspace and a window that lets you look into the *Customer* table is created on the desktop. You can change the attributes of the window on the desktop—such as size, location, and so forth—without changing the workspace. If you coedit the *Customer* table, however, you are changing the workspace; coediting actions do not affect the desktop.

---

## Floating windows

The optional keyword `FLOATING` lets you create a window in the application layer. For example, you can issue the command:

```
WINDOW CREATE FLOATING TO HelpWindow
```

Windows that you create with the `FLOATING` keyword are above the echo layer; these windows are always visible even if `ECHO` is `OFF`. Because floating windows are canvas windows, they are useful for displaying messages or help to the user. For example, a floating window could display a message such as “Performing a query” while you open a table on the workspace and perform a query with `ECHO OFF`.



Floating windows are intended for use with ECHO OFF. Because floating windows are above the echo layer, they are a convenient way to display a message while an ECHO OFF hides workspace operations. If you attempt to use floating windows with ECHO NORMAL, you will see unusual behavior. For example, if a floating window is onscreen with a workspace window while ECHO is NORMAL, you can select the workspace window but it will not rise above the floating window in the z-order—application layer windows always remain above desktop layer windows in the z-order.

You can move a canvas window between the application layer and the desktop layer by changing the value of the FLOATING attribute. The FLOATING attribute is set to False for windows in the desktop layer and set to True for windows in the application layer. See “Working with window attributes” earlier in this chapter for a description of changing attributes. Floating windows are available only while an application is running; they are closed automatically at the end of an application.

---

## Floating windows and desktop objects

When you create a floating window, the entire window group above the echo layer becomes current. You can view a table or create a canvas window in the desktop after you create a floating window, but the last floating window you created is still current; GETWINDOW() and WINDOW HANDLE CURRENT will interact with that floating window. The floating window group will remain current until you explicitly select a window in the desktop layer. Example 11-1 demonstrates the interaction between floating windows and desktop objects.

---

### Example 11-1 Floating window and desktop interactions

This script creates a floating window and displays its handle with a MESSAGE statement. The script then places a table on the workspace and uses GETWINDOW() to get the handle of the current window. Because the desktop window has not been explicitly selected, GETWINDOW() still returns the handle of the floating window. When the desktop window is explicitly selected with a WINDOW SELECT command, the desktop window becomes the current window and GETWINDOW() returns its handle.

```
WINDOW CREATE FLOATING TO FloatWin ; creates a floating window
MESSAGE GETWINDOW()                ; displays handle of the floating window
SLEEP 2000
MESSAGE ""                          ; clears the message

VIEW "Customer"                    ; views Customer table
WINDOW HANDLE IMAGE 1 to TableWin
MESSAGE "The customer table is on the workspace"
SLEEP 1000
MESSAGE GETWINDOW()                ; still displays handle of floating window
SLEEP 2000
MESSAGE ""                          ; clears the message

WINDOW SELECT TableWin              ; explicitly selects the table
```

```
MESSAGE "The customer table is now selected"  
MESSAGE GETWINDOW() ; now displays handle of the table  
SLEEP 2000
```

In Example 11-1, the table window does not rise above the floating window when it is selected; as mentioned previously, desktop windows cannot rise above floating windows in the z-order. When the table window is selected with the WINDOW SELECT command, the floating window appears *deselected* onscreen. Because echo is off, the user cannot see the table window; however, the floating window remains visible and appears deselected to the user.

While floating windows are displayed, you can view and manipulate desktop windows *without affecting the selection state* of floating windows. You should avoid explicitly selecting desktop windows while floating windows are visible unless you intentionally want to create an unusual effect. Chapter 12, "Controlling screen display," discusses echo and its effect on windows.

# Controlling screen display

PAL canvases are used to display output to a user. To give you complete control over the display, there are three types of canvases for use in different situations. Paradox lets you control exactly how much of your application the user sees. You will frequently want to work with objects, such as Lookup tables, that you alternately want to hide and reveal. Other times, you will want to show objects to users, but control their appearance on the screen. By understanding canvases and knowing how to control your system's echo, you can control what the user sees onscreen while you manipulate Paradox. This chapter shows you how to use PAL to control the display of the screen.

This chapter covers

- working with canvases
- controlling the display of the desktop
- specifying the current canvas
- writing on a canvas

---

## Working with PAL canvases

PAL canvases are used to display output to a user. There are three types of PAL canvases:

- canvas windows that are explicitly created with the WINDOW CREATE command
- canvases that are associated with Paradox objects such as table windows
- the full-screen canvas

---

## Canvas windows

Canvas windows are windows that are explicitly created with the WINDOW CREATE command. Canvas windows contain only a canvas and do not contain Paradox objects such as tables or queries. WINDOW CREATE lets you use optional keywords to specify the size, location, and attributes of canvas windows you create. For complete information about creating and manipulating canvas windows, see Chapter 11, “Using windows.”

---

## Paradox object canvases

An image window—a window that contains a Paradox object such as a table or a query—can also display a canvas. When an image window displays its canvas, you can write a message directly on it. Writing on an image window’s canvas in this manner is called *annotating* an object. You can cause an image window to display a canvas by using the SETCANVAS command. See “Specifying the current canvas” later in this chapter for information about using the SETCANVAS command.

---

## The full-screen canvas

The full-screen canvas is the size of the entire screen. As shown in Figure 11-3 in Chapter 11, the full-screen canvas sits at the top of the echo layer. If you do not specify a canvas to display output to a user, the full-screen canvas is used as the default canvas. The canvas used in versions of Paradox prior to 4.0 was the full-screen canvas. See “Specifying the current canvas” later in this chapter for information about writing to the full-screen canvas.

---

## Showing the user the desktop

When a script begins running, PAL automatically issues the command ECHO OFF. When ECHO OFF is issued (either automatically at the beginning of the script or explicitly by you), the screen display is “frozen.” A picture of the existing desktop state is placed on the full-screen canvas. The desktop display is not updated again until ECHO NORMAL, ECHO SLOW, or ECHO FAST is issued. Changes to canvas windows and desktop objects can occur while echo is off, but these changes cannot be seen until you “echo” the display of the desktop to the screen again with ECHO NORMAL.

Because the ECHO command controls the display of the entire desktop, it is often called *global* echo. You can also control *local* echo—the display of changes within a single image window—with the WINDOW ECHO command. The WINDOW ECHO command is discussed in “Controlling local echo,” later in this chapter.

With the ECHO command, you can control the display of the desktop by either freezing its display on the full-screen canvas or revealing it,

depending on what command is being executed. The ECHO command is particularly useful for debugging and demonstration. You can embed ECHO commands in various places in a script to alternately reveal the desktop and conceal operations that affect the desktop.

For example, to provide a snapshot of the canvas at a particular point in a PAL script, issue the statements:

```
ECHO NORMAL
ECHO OFF
```

To keep a message displayed onscreen while a query that the user cannot see is performed on the desktop, you can use the following:

```
WINDOW CREATE TO DisplayWindow
? "Performing query..."
ECHO NORMAL
ECHO OFF
QUERY          ; the query creates a window that is hidden from
...           ; the user because echo is off
ENDQUERY
DO IT!
...
```

You can also simply leave echo off to obscure the query and use a floating canvas to display a message above the echo layer, as follows:

```
ECHO OFF
WINDOW CREATE FLOATING TO DisplayWindow
? "Performing query..."
QUERY          ; the query creates a window that is hidden from
...           ; the user because echo is off
ENDQUERY
DO IT!
...
```

See Chapter 11, "Using windows," for a complete description of floating canvases.

---

## Controlling local echo

The WINDOW ECHO command controls local echo—the display of changes within an individual image window. When an image window is opened, local echo is True by default. If local echo is True, changes to the window will be shown in that window. If local echo is False, changes to the window will not be shown in the window.

Use the window echo command by specifying a window handle and a value of logical True or False:

```
ECHO NORMAL
COEDIT "Customer"          ; place Customer table on workspace
CustomerWindow = GETWINDOW() ; gets a handle for Customer window
WINDOW ECHO CustomerWindow False ; hides changes to Customer window
```

```
END ; moves cursor to last record of table
K = GETCHAR() ; new cursor location now visible
```

If global echo is normal, the entire desktop is revealed to the user; however, you can continue to control the display of changes that affect individual image windows by changing the local echo of those windows. If global echo is off, no changes will be visible, but local echo will still control what becomes visible when global echo is turned back on.

Because local echo is a window attribute, you can specify the ECHO attribute in a dynamic array and then use WINDOW SETATTRIBUTES to set the echo state for the window, as described in Chapter 11, "Using windows."

Do not confuse the ECHO attribute and the CANVAS attribute of windows. The ECHO attribute applies only to image windows, and it controls whether changes to the workspace that affect an image window are echoed to the screen. The CANVAS attribute affects only a canvas window or the canvas of an image window, and it controls whether changes to that canvas are displayed immediately or later.

WINDOW ECHO is always False for windows created by WINDOW CREATE since they do not have an attached Paradox object they can possibly display. Issuing a SETCANVAS command to an image window causes the image window to display its attached canvas and implies a WINDOW ECHO False. Setting WINDOW ECHO to True for an image window that is displaying a canvas causes the canvas to be discarded.

---

## Specifying the current canvas

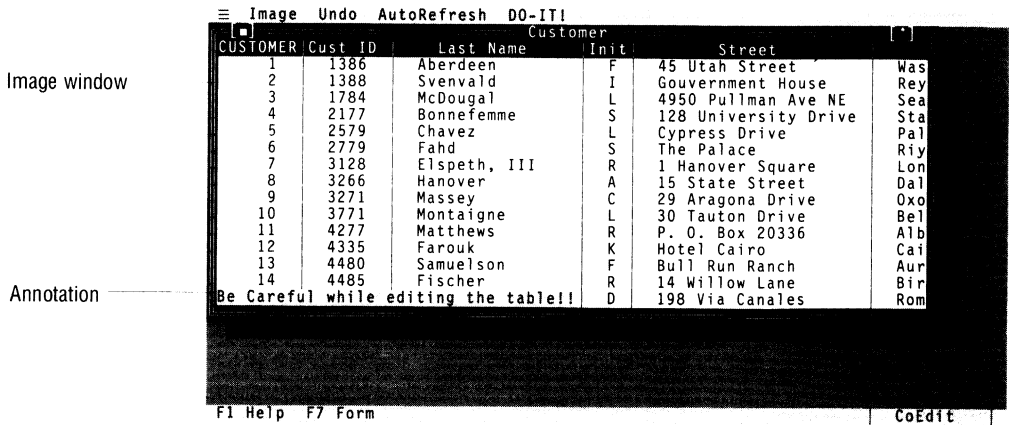
The *current* canvas is the canvas that is used to display the output of canvas-writing commands such as ? and ??.

When a PAL application starts running, the current canvas is the full-screen canvas. You can specify a different current canvas by issuing the command SETCANVAS *WindowHandle*. You can also restore the full-screen canvas to being the current canvas at any time by issuing the command SETCANVAS DEFAULT.

As shown in Figure 12-1, the following example explicitly specifies *CustomerWindow* as the current canvas and annotates it:

```
COEDIT "Customer" ; places Customer on desktop in CoEdit mode
CustomerWindow = GETWINDOW() ; gets a handle for Customer
SETCANVAS CustomerWindow ; makes Customer the current canvas
STYLE ATTRIBUTE 15 + 64 ; color text white on red
@15,0
?? "Be Careful while editing the table!!"
ECHO NORMAL ; show the text on screen
SLEEP 5000
```

Figure 12-1 Annotating an image window



When you create a canvas window with the `WINDOW CREATE` command, the current canvas is automatically set to that window. For example, even though the previous script explicitly specified a current canvas, a later statement such as:

```
WINDOW CREATE TO NewWindow
```

will implicitly make *NewWindow* the current canvas.

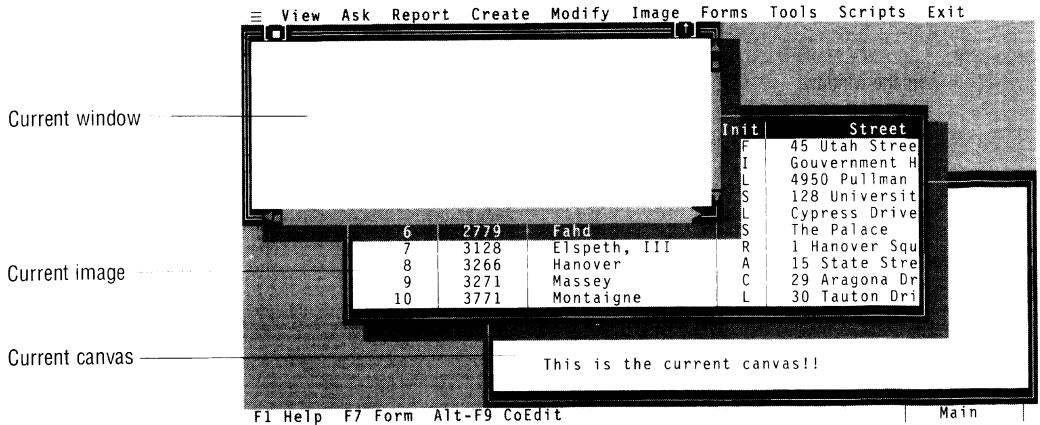
If you have not explicitly specified a current canvas with the `SETCANVAS` command or implicitly specified a current canvas with `WINDOW CREATE`, a canvas-writing command will affect the full-screen canvas. If the current canvas window has been closed, a canvas-writing command will result in a script error.

Do not confuse the current canvas with the current window or the current image. The current window is the one that appears above other windows onscreen; the current image is the Paradox object that was most recently the current window. The current canvas is not necessarily the same as either the current image or the current window. You can have a different current canvas, current window, and current image onscreen at the same time. For example,

```
WINDOW CREATE TO DisplayWindow
VIEW "Customer"           ; the current image
WINDOW CREATE TO NewWindow ; the current window
SETCANVAS DisplayWindow   ; the current canvas
```

Figure 12-2 illustrates the difference between the current canvas, the current image, and the current window.

Figure 12-2 Current canvas, current image, and current window



For further information about the current window and the current image, see the section called "Specifying the current window" in Chapter 11, "Using windows."

The GETCANVAS() function returns the window handle of the current canvas. For example, the following code demonstrates that a window created with the WINDOW CREATE command is the current canvas:

```
WINDOW CREATE TO MyCanvas
MESSAGE MyCanvas      ; displays the handle of MyCanvas
SLEEP 2000
MESSAGE ""           ; clears the message display
SLEEP 2000
MESSAGE GETCANVAS() ; displays the same handle
SLEEP 2000
MESSAGE ""
```

Because a user can close a window, you should always check to make sure that the current canvas is still available before attempting to write to it. The following code uses the GETCANVAS() and ISWINDOW() functions to determine if the current canvas is available before writing to it:

```
IF ISWINDOW(GETCANVAS()) ; make sure the current canvas is available
    THEN ? "The current canvas is alive and well"
ENDIF
```

To prevent a subordinate procedure from changing the current canvas in the calling procedure, use GETCANVAS() at the beginning of the subordinate procedure to store the current canvas and then restore the calling procedure's current canvas at the end of the subordinate procedure:

```
PROC Watch() ; begins the procedure definition
    PRIVATE OldCanvas ; prevents variable from affecting global
```



```

                                ; variable with same name
01dCanvas = GETCANVAS()        ; saves the current canvas to 01dCanvas
WINDOW CREATE FLOATING @10,26  WIDTH 30 HEIGHT 5 TO ProcCanvas
@1,2
?? "Watch out for that tree!"
SLEEP 2000
WINDOW CLOSE                   ; close the window we just opened
SETCANVAS 01dCanvas           ; restores the current canvas at end of proc
ENDPROC
Watch()

```

---

## Turning the canvas off and on

When you are on a canvas for an extended period of time, you might want to construct the entire presentation before showing it to the user. The CANVAS command is used to control whether changes to the current canvas are displayed onscreen immediately or later. When used with the OFF option, the CANVAS command lets you write on a canvas without immediately showing the writing to the user. You can build the canvas part by part and then reveal it all at once by issuing CANVAS ON.

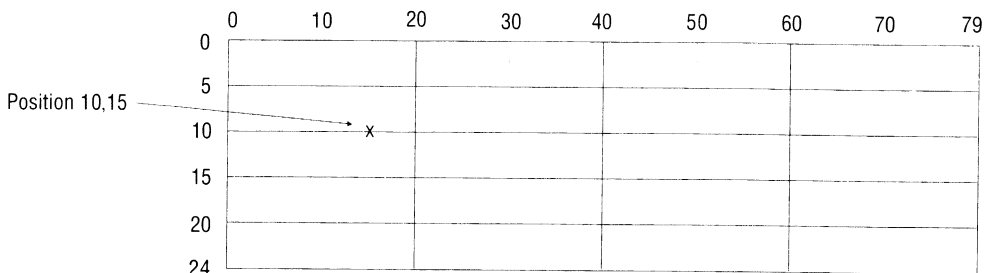
As mentioned previously, CANVAS is also a window attribute. You can specify the CANVAS attribute in a dynamic array and then use WINDOW SETATTRIBUTES to set the CANVAS state for a window, as described in Chapter 11, "Using windows."

---

## Writing on a canvas

The canvas cursor specifies the location where information will be placed on a canvas. Locations on a canvas are specified relative to the upper left corner of the canvas, not the screen. Canvas coordinates start with row 0, column 0, not with row 1, column 1. If a canvas has a frame, the frame is not included as a possible location. Figure 12-3 illustrates canvas coordinates.

Figure 12-3 Canvas coordinates



---

## Positioning the canvas cursor

The @ command sets the location of the canvas cursor in rows and columns relative to the upper left corner of the current canvas. For example, the command @3, 9 places the canvas cursor on the fourth row and the tenth column of the current canvas. The default location of the canvas cursor is the first row and first column of the current canvas, which has the coordinates 0, 0.

The row and column numbers specified in an @ statement or within a canvas-writing command must be less than the total height and width of that canvas. If you create a framed canvas window, the height and width that you specify include the frame. For example, when you issue the command:

```
WINDOW CREATE HEIGHT 10 WIDTH 20 TO NewCanvas
```

you create a window with an 8-row, 18-column canvas. If you try to move the cursor with the command @8, 18, you will get an error.

If you lose track of the PAL cursor, you can use the ROW() and COL() functions to find out where the cursor is. These functions return integer values which respectively represent the row and column position of the cursor on the current canvas. For example, if you have moved the PAL cursor to position 10, 15, these commands capture that position into the variables *RowVal* and *ColVal*:

```
@10,15          ; position the cursor
RowVal = ROW()  ; assigns 10
ColVal = COL()  ; assigns 15
```

ROW() and COL() always return the position of the cursor on the current canvas, and not the position of the Paradox workspace cursor (the cursor position in the current image). While there is no way to directly determine the position of the Paradox cursor on the workspace, you can use the SYNCCURSOR command to synchronize the locations of the canvas cursor and the workspace cursor so you can annotate an image window. See the *PAL Reference* for complete information about the SYNCCURSOR command.

Finally, there are times when you want to change the style of the canvas cursor or when you don't want the canvas cursor to be displayed at all. You can use the CURSOR command to do this:

```
CURSOR OFF      ; hides the cursor
CURSOR NORMAL   ; redisplay the cursor
```

---

## Displaying messages and text

There are several ways of displaying information on the PAL canvas. PAL has commands and functions that write simple messages, multi-line messages, formatted messages, and messages in various colors, styles, and display attributes (bold, reverse video, and so on).

The simplest writing commands are the ? and ?? commands. Both of these commands take an expression or list of expressions and display

their values on the current canvas. The ? command displays the values starting at the beginning of the line below the cursor, while ?? displays them starting at the cursor position. Both display the values as strings in the current display attribute (more on that later in this chapter). After the commands are executed, the cursor will be in the canvas position following the last value displayed.

For example, the script

```
WINDOW CREATE @1,0 HEIGHT 23 WIDTH 80 TO DisplayWindow
@ 10,15           ; position cursor
? "Today is ", TODAY() ; show today's date
RowVal = ROW()   ; get row position of cursor
ColVal = COL()   ; get column position
```

positions the cursor on the 11th line and 16th column (because the ? command moves the cursor down one line) and displays a message like **Today is 1/12/92**. At the completion of this script, the cursor has moved to the end of the date, so for the date value 1/12/92 *RowVal* contains 11 and *ColVal* contains 31. If we had used ?? instead of ?, the message would have been displayed on the 10th line and *RowVal* would contain 10 and *ColVal* 16.

If you want to put several lines of literal text on the PAL canvas, use the TEXT command instead of ? or ?. TEXT displays multiple lines of text beginning at the current cursor position. TEXT cannot contain any expressions, only text characters. After the text is displayed, the cursor will be at the beginning of the line following the last line of text. You can use the SETMARGIN command or the MARGIN window attribute to establish a left margin for blocks of text you display with the TEXT, ?, and ?? commands.

---

## Formatting values

The FORMAT() function (described in Chapter 5 and the *PAL Reference*) formats values for display. With FORMAT(), you can specify

- the width of the value as displayed (number of columns it takes up)
- the alignment of the information within the area (left, right, or centered)
- the case of letters in the value (upper, lower, initial caps)
- dollar signs, commas, leading zeros, and other characters to be included in displaying numeric values
- the format of dates and logical values

FORMAT() does not actually display anything by itself. It must be used with the ? and ?? commands already discussed, or with the MESSAGE command. For example, if you want to display the date in dd-Mon-yy format, you would use this command instead of the simpler ? command in the sample script presented in the last section:

```
@ 10,15 ; position the cursor  
?? "Today is ", FORMAT("D7",TODAY())
```

Although you don't need to use `FORMAT()` to display values, it does give you more control over how things appear on the canvas. As you gain experience in creating scripts, you'll probably find that you use complex `FORMAT()` statements much more often than simple `?` or `??` commands with text expressions.

---

## Setting canvas colors and styles

The `PAINTCANVAS` and `STYLE` commands control the colors on a PAL canvas. You can also use these commands to control other video attributes such as reverse video or high intensity on a monochrome display. For color displays, you can use the standard palette of 8 background and 16 foreground colors. Appendix F of the *PAL Reference* lists the codes you need to produce each combination of background/foreground colors.

The major difference between the `PAINTCANVAS` command and the `STYLE` command is that one works after the fact and the other works before the fact. The `PAINTCANVAS` command lets you paint color on existing text. The `STYLE` command, on the other hand, applies to everything you subsequently write on a particular canvas until you change styles by issuing another `STYLE` command.

If you want to harmonize the colors you paint on the PAL canvas with those currently being used on the Paradox workspace, you can use the `SYSCOLOR()` function or the `GETCOLORS` command to determine the currently set Paradox system colors. `SYSCOLOR()` and `GETCOLORS` are described in "Setting system colors," later in this chapter. When you use `STYLE` and `PAINTCANVAS`, keep in mind that monochrome and color systems require different handling. For example, you can't underline text on a color monitor and you can't set colors on a monochrome system. Thus, you should generally use the `MONITOR()` function to determine what kind of monitor the script is playing on before issuing any `STYLE` or `PAINTCANVAS` commands.

---

### STYLE

When a script begins playing, the default style is determined by the preferences established for the Paradox workspace. To display something in a different style, use the `STYLE` command prior to painting it. For example, to display blue text on a black background (or to underline something on a monochrome screen), use

```
STYLE ATTRIBUTE 1
```

before using `?`, `??`, or `TEXT` to display it. A complete description of all the options, including attribute codes, available with `STYLE` is included in the *PAL Reference*. When you want to return to the default display, use `STYLE` alone (without options).

As mentioned previously, each `STYLE` command applies to everything you subsequently paint on the canvas—until the next `STYLE` command. When you switch styles, anything already on the canvas remains in the style in which it was originally painted.

---

## PAINTCANVAS

The `PAINTCANVAS` command lets you apply an arbitrary background/foreground color combination to an area or border on the canvas. In the command, you specify the color attribute you want to apply and the cursor coordinates defining the corners of the area or border. For example, if you want to display a block of white text on a blue background surrounded by a magenta border, use the following code fragment:

```
WINDOW CREATE @3,4 WIDTH 70 HEIGHT 8 TO DisplayWindow
SETMARGIN 6
@2,0
TEXT
This is some text that is going to be displayed on the
PAL canvas as a demonstration of the PAINTCANVAS command.
ENDTEXT
PAINTCANVAS ATTRIBUTE 31 0,0,5,67
PAINTCANVAS BORDER ATTRIBUTE 80 0,0,5,67
```

---

## Setting global and local colors

`Paradox` lets you examine and change the color attributes of your system (global colors) and of individual windows (local colors). Global colors are controlled by the `GETCOLORS` and `SETCOLORS` commands and the `SYSCOLOR()` function; local colors are controlled by the `WINDOW GETCOLORS` and `WINDOW SETCOLORS` commands. `GETCOLORS`, `WINDOW GETCOLORS`, and `SYSCOLOR()` are used to determine the current color attributes; `SETCOLORS` and `WINDOW SETCOLORS` are used to change color attributes.

---

### Global colors

When you specify global colors you can establish the color attributes for your entire system at the same time. Global colors affect all windows (as well as other interface objects); however, you can also specify local window colors that have priority over global color settings.

---

#### *The `GETCOLORS` command*

The `GETCOLORS` command reads the color attributes of the current palette into a dynamic array. The index of each element in the

dynamic array represents a specific attribute of the current palette. The value of each element is a number that controls the color of the index. Appendix F in the *PAL Reference* describes how to determine the colors represented by the value of each element.

The dynamic array created by GETCOLORS uses the indexes 0–31 and 1000–1077 to represent colors that are now used in Paradox; additional indexes may be added to later versions. The indexes 0–31 were available in versions of Paradox prior to 4.0 and identify the same attributes in Paradox 4.0; however, some of these attributes are now meaningful in compatible mode only. Indexes 0–31 are described in Table 12-1; attributes meaningful in compatible mode only are indicated with an asterisk (\*) in this table.

Table 12-1 Attributes of screen elements 0–31

Index	Element	Description
<b>General</b>		
0*	Menu background	Top two screen lines and menu choices
2*	Current menu selection	Current menu choice
4*	Menu selection description	Annotation line that describes the current menu choice
5*	Mode indicator	Mode indicator at the top right corner of the screen
7	Selected area	Area selection in the Form Designer and Editor
8	Workspace area	Empty workspace areas in compatible mode; Editor, Report Designer, canvas window background, and default background text in standard mode
19	Examples and checkmarks	Checkmarks and example elements in a query image
<b>Image display</b>		
6*	Noncurrent table border	Non-current image(s) in the workspace
9	Current table border	Current table in compatible mode; any table or query windows in standard mode
<b>Field display</b>		
10	Reserved	Reserved for system use
11	Reserved	Reserved for system use
17	Field values	Field values in table view, queries, and Report Designer

<b>Index</b>	<b>Element</b>	<b>Description</b>
18	Negative values	Negative values in numeric or currency fields
<b>Message/Status display</b>		
3	Message value and PAL Debugger line	Messages in bottom right corner; current line in PAL Debugger
16	PAL Debugger status line	Status of PAL Debugger
<b>Help/System Forms</b>		
12	Border	Help and system forms border
13	Text	Help and system forms text
14	Help text	Highlighted help text
15	Help text	Reverse video text
21	Help index text	Normal help index text
22	Help index text	Horizontal lines of the help index
23	Help index text	Current selection of the help index
<b>Report</b>		
24	Vertical ruler	Vertical ruler and page width edge marker in Report Designer
25	Report band lines	Band lines in the Report Designer
<b>Form</b>		
27	Reverse video and high intensity	High intensity and reverse video mapping in the Form Designer on B&W monitors
28	Reverse video text	Reverse video mapping in the Form Designer on B&W monitors
29	High intensity	High intensity mapping in the Form Designer on B&W monitors
<b>Reserved</b>		
1	Undefined	Reserved for future use
20	Undefined	Reserved for future use
26	Undefined	Reserved for future use
30	Undefined	Reserved for future use
31	Undefined	Reserved for future use

\* These elements are only meaningful in compatible mode

The indexes 1000–1077 were added to represent the new screen elements of Paradox 4.0. These indexes are described in Table 12-2.

Table 12-2 Attributes of screen elements 1000–1077

<b>Index</b>	<b>Element</b>	<b>Description</b>
<b>General</b>		
1000	Desktop	Desktop area
<b>Menu</b>		
1001	Text	Normal menu command
1002	Text	Disabled menu command
1003	Text	Hot key for menu command
1004	Text	Highlighted menu command
1005	Text	Highlighted disabled menu command
1006	Text	Hot key for highlighted menu command
<b>Windows</b>		
1007	Frame	Inactive window frame
1008	Frame	Active window frame and title
1009	Frame	Frame icons (such as the close box) and selected frame (when dragging)
1010	Scroll bar	Scroll bar
1011	Scroll bar	Scroll bar controls
1012	Undefined	Reserved for future use
1013	Undefined	Reserved for future use
1014	Undefined	Reserved for future use
<b>Reserved</b>		
1015-1030	Undefined	Reserved for future use
<b>Dialog boxes</b>		
1031	Frame	Inactive dialog box frame
1032	Frame	Active dialog box frame and title
1033	Frame	Selected dialog box frame (when dragging); icons if present
1034	Scroll bar	Scroll bar
1035	Scroll bar	Scroll bar controls
1036	Text	Default background text (effective after REPAINTDIALOG)
1037	Label	Label when linked control is inactive
1038	Label	Label when linked control is active
1039	Label	Label hot key
1040	Push button	Text for normal push button label
1041	Push button	Text for default push button label



<b>Index</b>	<b>Element</b>	<b>Description</b>
1042	Push button	Text for selected push button label
1043	Push button	Reserved for future use
1044	Push button	Hot key for push button label
1045	Push button	Button shadow
1046	Cluster element	Normal radio button or check box item
1047	Cluster element	Highlighted radio button or check box item
1048	Cluster element	Hot key for radio button or check box
1049	Type-in box	Normal type-in box text
1050	Type-in box	Selected type-in box text
1051	Type-in box	Type-in box arrows
1052	Undefined	Reserved for future use
1053	Undefined	Reserved for future use
1054	Undefined	Reserved for future use
1055	Undefined	Reserved for future use
1056	Pick list	Normal pick list item text
1057	Pick list	Highlighted pick list item text when pick list is active
1058	Pick list	Selected pick list item text when pick list is not active
1059	Pick list	Column dividers
1060	Undefined	Reserved for future use
1061	Undefined	Reserved for future use
1062	Undefined	Reserved for future use

### ***Miscellaneous***

1063	Speed bar	Normal item text
1064	Speed bar	Reserved for future use
1065	Speed bar	Hot key for item
1066	Speed bar	Selected item
1067	Speed bar	Reserved for future use
1068	Speed bar	Hot key for selected item
1069	Speed bar	Vertical separator line
1070	SQL title	SQL server title for a selected replica table
1071	Speed bar	Menu prompts and status text

### ***Paradox Application Workshop***

1072	AltMenu	Normal menu command
1073	Undefined	Reserved for future use
1074	AltMenu	Hot key for menu command

Index	Element	Description
1075	AltMenu	Highlighted menu command
1076	Undefined	Reserved for future use
1077	AltMenu	Hot key for highlighted menu command

For example, the index 1000 is used to represent the desktop. To determine what color the current palette uses for that element, you could enter the following:

```
GETCOLORS TO ColorList ; creates a dynamic array
RETURN ColorList[1000] ; returns the value of the color
```

The GETCOLORS command is most useful when you want to determine the complete set of colors in the current palette. To determine an individual color, you could also use the SYSCOLOR() function as follows:

```
RETURN SYSCOLOR(1000)
```

If you have not changed your default palette, the color value returned by the last statement for color systems is 113. In Appendix F of the *PAL Reference*, you see that the color value 113 is composed of 1 plus 112. Because 1 represents blue and 112 represents light gray, the color used by the default palette for the desktop is blue on a light gray background.

---

### **The SETCOLORS command**

The SETCOLORS command lets you set any or all of the attributes of the current palette from a dynamic array. Like GETCOLORS, the index of each element represents a specific attribute of the current palette, and the value of each element is a number that controls the attribute.

The SETCOLORS dynamic array uses the same indexes that are returned by the GETCOLORS command. Any element that you do not specify for use with the SETCOLORS command remains unchanged. For example, to change the color of the desktop to red on a light gray background without changing other elements you would enter the following:

```
DYNARRAY ColorSettings[] ; create a dynamic array
ColorSettings[1000] = 116 ; make desktop red on light gray
SETCOLORS FROM ColorSettings ; set the colors
```

Appendix F of the *PAL Reference* shows that the color value 116 is composed of 4 plus 112. Because 4 represents red and 112 represents light gray, the color now used by the current palette for the desktop is red on a light gray background.

---

### **The SYSCOLOR() function**

SYSCOLOR() takes an argument that is an integer expression between 0 and 31 or between 1000 and 1077. The integer represents a

specific Paradox screen element. The value that SYSCOLOR() returns represents the color of that screen element.

SYSCOLOR() references the same screen elements as GETCOLORS and SETCOLORS. The screen elements 0–31 were available in versions of Paradox prior to 4.0 and are the same in Paradox 4.0; however, some of these elements are now meaningful in compatible mode only. The elements 1000–1077 were added to represent the new screen elements of Paradox 4.0. Elements 0–31 are described in Table 12-1; attributes meaningful in compatible mode only are indicated with an asterisk in this table. Elements 1000–1077 are described in Table 12-2.

To determine the color used by a highlighted menu selection in compatible mode (element 2), enter the statement:

```
RETURN SYSCOLOR(2) ; returns 31 for the default color palette
```

This is the same value that is returned by the following:

```
GETCOLORS TO ColorList  
RETURN ColorList[2]
```

The SYSCOLOR() function is slightly faster than the GETCOLORS command because SYSCOLOR only references a single element. GETCOLORS references all the elements, even if you need only one.

---

## Local colors

When you specify global colors, you are setting the attributes for the following types of windows:

- active and inactive images
- active and inactive canvases
- active and inactive dialog boxes

The global colors result in each type of window having the same attributes. For example, all active image windows look alike; similarly, all inactive canvases are the same.

When you use local color palettes, you can control the color attributes of any individual window. Local colors have priority over global colors and affect only the windows that you specify. You can use different colors for as many individual windows as you want.

---

## **The WINDOW GETCOLORS command**

WINDOW GETCOLORS reads the color attributes of the local palette for a specified window into a dynamic array. Like the dynamic array created by the GETCOLORS command, the index of each element in this dynamic array represents a specific attribute of the local palette; the value of each element is a number that controls the color of the index.

To use WINDOW GETCOLORS, specify the handle of the window you want the color attributes for. See Chapter 11, "Using windows," for information about getting window handles.

The following code sample creates a dynamic array representing the attributes of a local color palette:

```
VIEW "Bookord"                                ; view a table
WINDOW HANDLE IMAGE 1 TO TableWindow          ; get a handle for the table window
WINDOW GETCOLORS TableWindow TO TableColors ; capture the local palette
```

Appendix F in the *PAL Reference* describes how to determine the colors represented by the value of each element. Table 12-3 describes the color attributes that are returned in the dynamic array created by WINDOW GETCOLORS for a window.

Table 12-3 Local color attributes for windows

Index	Element	Description
0	Frame	Inactive window frame
1	Frame	Active window frame and title
2	Frame	Frame icons (such as the close box) and selected frame (when dragging)
3	Scroll bar	Scroll bar
4	Scroll bar	Scroll bar controls
5	Undefined	Reserved for future use
6	Undefined	Reserved for future use
7	Undefined	Reserved for future use

Table 12-4 describes the color attributes that are returned in the dynamic array created by WINDOW GETCOLORS for a dialog box.

Table 12-4 Local color attributes for dialog boxes

Index	Element	Description
0	Frame	Inactive dialog box frame
1	Frame	Active dialog box frame and title
2	Frame	Selected dialog box frame (when dragging); icons if present
3	Scroll bar	Scroll bar
4	Scroll bar	Scroll bar controls
5	Text	Default background text
6	Label	Label when linked control is inactive
7	Label	Label when linked control is active
8	Label	Label hot key
9	Push button	Text for normal push button label

Index	Element	Description
10	Push button	Text for default push button label
11	Push button	Text for selected push button label
12	Push button	Reserved for future use
13	Push button	Hot key for push button label
14	Push button	Button shadow
15	Cluster element	Normal radio button or check box item
16	Cluster element	Highlighted radio button or check box item
17	Cluster element	Hot key for radio button or check box
18	Type-in box	Normal type-in box text
19	Type-in box	Selected type-in box text
20	Type-in box	Type-in box arrows
21	Undefined	Reserved for future use
22	Undefined	Reserved for future use
23	Undefined	Reserved for future use
24	Undefined	Reserved for future use
25	Pick list	Normal pick list item text
26	Pick list	Highlighted pick list item text when pick list is active
27	Pick list	Selected pick list item text when pick list is not active
28	Pick list	Column dividers
29	Undefined	Reserved for future use
30	Undefined	Reserved for future use
31	Undefined	Reserved for future use

The elements used to describe local color attributes are the same as the elements that describe global color attributes, but the indexes of the local color elements are offset from the global color elements by a fixed amount. For windows, the local color indexes 0–7 are offset from the global color indexes 1007–1014 by 1007. For dialog boxes, the local color indexes 0–31 are offset from the global color indexes 1031–1062 by 1031.

Because the local and global color indexes are related in this manner, you can easily convert between global and local colors. For example, the following script uses GETCOLORS to create a dynamic array of global colors and then offsets the indexes 1031 through 1062 to create a dynamic array for local colors:

```

GETCOLORS TO GlobalColors           ; dynamic array for global colors
DYNARRAY LocalColors[]             ; dynamic array for local colors
FOR i FROM 1031 TO 1062             ; the global dialog box indexes
  LocalColors[i-1031] = GlobalColors[i] ; local indexes 0-31 now created
ENDFOR                             ; from global indexes 1031-1062

```

---

## The **WINDOW SETCOLORS** command

The **WINDOW SETCOLORS** command lets you set the local color attributes of any window from a dynamic array. Like **WINDOW GETCOLORS**, the index of each element represents a specific attribute of the local palette, and the value of each element is a video attribute.

The **WINDOW SETCOLORS** dynamic array uses the same indexes that are returned by the **WINDOW GETCOLORS** command. For example, to change the color of a canvas window you would enter the following:

```
DYNARRAY CanvasColorSettings[] ; create a dynamic array
CanvasColorSettings[0] = 127 ; make inactive frame white on light gray
CanvasColorSettings[1] = 112 ; make active frame black on light gray
WINDOW CREATE TO CanvasWindow ; create a canvas window
WINDOW CREATE TO OtherWindow ; create another canvas window
WINDOW SETCOLORS CanvasWindow FROM CanvasColorSettings
```

After running the above script, you can see that the frame of *CanvasWindow* is white on light gray when *OtherWindow* is active; it is black on light gray when *CanvasWindow* is active.

To change the color of an image window or a dialog box, first get the handle for the window or dialog box:

```
DYNARRAY ImageColorSettings[] ; creates a dynamic array
ImageColorSettings[0] = 127 ; make inactive frame white on light gray
ImageColorSettings[1] = 112 ; make active frame black on light gray
VIEW "Bookord" ; open the Bookord table
WINDOW HANDLE IMAGE 1 TO ImageWindow ; get a handle for Bookord
WINDOW SETCOLORS ImageWindow FROM ImageColorSettings
```

After you change the local colors, you can make any window reflect the settings of the global palette again by using the **DEFAULT** keyword, as follows:

```
WINDOW SETCOLORS ImageWindow DEFAULT
```

# Handling events

This chapter describes the event-driven programming model of Paradox. Every user interaction with Paradox generates an event. PAL lets you examine these events to determine what the user wants to do and to control your application.

This chapter covers

- events and their categories
- the event stream
- event trapping and blocking
- event processing in applications
- valid events
- triggers

For your convenience when using the event-processing commands in Paradox, the information in this chapter is summarized in Appendix J of the *PAL Reference*.

---

## What is an event?

An event is an individual packet of information that completely describes a specific, discrete occurrence at a specific time. Each user keystroke or mouse action generates an event; in addition, certain other conditions also generate an event. Events cannot be broken down into smaller pieces; a user typing a word is not a single event, but a series of individual key events.

Paradox recognizes four categories of events:

- mouse events, generated by an individual mouse action such as a movement

- key events, generated by an individual key action such as pressing *Ctrl-F9*, *a*, or *Shift-F6*
- message events, generated by your system on attempting an interaction with a window or a SHOWPULLDOWN menu
- idle events, generated at regular intervals by your system

In addition to events, Paradox uses triggers to describe specific interactions that occur in certain situations. See “Using triggers” later in this chapter for a complete description of triggers.

## The categories of events

Mouse and key events are each generated by a different physical interaction that occurs *externally* (outside your system): the former, by interactions with a mouse; the latter, by interactions with a keyboard. Message and idle events, however, are generated *internally* by your system. The following sections describe each event category.

### Mouse events

Mouse events originate externally and are generated whenever a user interacts with the mouse. There are four basic mouse events, each resulting from a different user interaction with the mouse. Table 13-1 describes the four types of mouse events.

Table 13-1 Mouse events

Mouse event	Caused by
UP	Releasing a mouse button
DOWN	Pressing a mouse button
MOVE	Moving the mouse
AUTO	Holding down the mouse without moving it (such as when scrolling)

### Key events

Key events originate externally and are generated whenever a user interacts with the keyboard. Pressing any alphanumeric or function key (and most other keys) generates a key event. *Ctrl*, *Alt*, and *Shift*, used in combination with another key, generate a different key event than simply pressing the key.

### Message events

Message events originate internally and are generated by the system when a user attempts a specific interaction; Paradox makes message events available to you during interactions with a window, or a SHOWPULLDOWN menu item or menu key. The PAL GETEVENT,



SHOWDIALOG, and WAIT commands can trap these message events and let you examine them.

There are five basic message events. Table 13-2 describes the three message events related to windows and the two message events related to SHOWPULLDOWN menus.

Table 13-2 Message events

Message event	Category	Meaning
CLOSE	Before	Close the current window
MAXIMIZE	Before	Maximize or restore the current window
NEXT	Before	Select the next window on the desktop
MENUSELECT	After	Menu item in a SHOWPULLDOWN menu has been selected
MENUKEY	After	Key in the UNTIL list of a SHOWPULLDOWN menu has been pressed

---

## Idle events

Idle events originate internally and are generated by your system at regular intervals when no other events are occurring. The frequency with which idle events are generated varies according to the speed of your system.

---

## The event stream

Events are queued for processing in the order in which they occur; the first event in the queue is processed first. Queued events move through your system in a “stream” that flows into your application; your application examines the event and decides either to handle it internally or to pass it on to Paradox for processing. Each event category is routed differently when it is executed.

---

## Mouse event routing

Because a mouse is a pointing device, a mouse event is only meaningful for a window or an object that is visible. When global echo is off, only floating windows and interface objects such as SHOWDIALOG dialog boxes and PAL menus are visible. No desktop windows are visible when echo is off. At most, only an *image* of a desktop window is visible on the full-screen canvas. See Chapter 12, “Controlling screen display,” for complete information about global echo.

A mouse event that occurs where no windows are visible is ignored; it is “thrown away” by Paradox. A mouse event that occurs where a window is visible, when executed, affects the interface object that is

visible directly beneath it. If a mouse event occurs above more than one interface object, the mouse event is meaningful to the object that is highest in the z-order. See Chapter 11, "Using windows," for complete information about the z-order.

Some mouse events, when executed, also result in a message event that is put into the queue at the top of the event stream. Only three mouse interactions generate message events that follow the mouse event: clicking the Close box (close message), double-clicking the title bar (maximize message), and clicking the Maximize/restore icon (maximize message).

---

## Key event routing

There are two kinds of key events: keys that are meaningful to windows (such as Maximize *Shift-F5*), and keys that are meaningful to Paradox objects. A key event that is meaningful to a window, when executed, also generates a message event that is put into the queue at the top of the event stream. A key event that is meaningful to a Paradox object, when executed, is passed on to the current Paradox object (such as a table or form).

A mouse click on the SpeedBar is also interpreted by Paradox as a key event, not as a mouse event. Because the SpeedBar only permits actions that are also available as Paradox key combinations, the mouse click does not generate a mouse event, only the key event that would be issued if that key combination were pressed. For example, clicking the mouse on HELP *F1* results in a key event, just like pressing the key *F1*.

---

## Message event routing

As mentioned previously, message events are issued as a result of certain mouse and key events and put into the queue at the top of the event stream. The message events are made available to PAL because they notify your application of interactions that are otherwise difficult to detect. Without message events, for example, it would be difficult to detect that a user clicked the Close box of a window unless you trapped for mouse down events and constantly localized and examined their coordinates.

All message events that are executed are routed to the current window. If no window is current, a message event is passed on to Paradox, where it is ignored. `MENUSELECT` and `MENUKEY` messages are also ignored by Paradox in this situation.

`MENUSELECT` and `MENUKEY` are *after* messages; they are significant because they indicate that a user has completed a specific type of interaction with a menu created by a `SHOWPULLDOWN` command. Your application can trap for `MENUSELECT` and `MENUKEY` messages so it can take the appropriate action when these messages occur.

Message events that occur *before* an action takes place can be *denied*. For example, a maximize message is generated when a user attempts to maximize a window. If your application is trapping for a maximize message, this message will be intercepted and not passed on to Paradox unless you decide to execute it. Table 13-2 lists the before and after messages. The next section discusses event trapping in detail.

---

## Trapping and blocking events

An *EventList* appears in the syntax of the GETEVENT, SHOWDIALOG, and WAIT commands as a comma-separated list of events. The *EventList* in these three commands acts as a filter; all the events in the *EventList* are *trapped* by your application and are not passed on to Paradox for processing unless you explicitly execute them using the EXECEVENT command.

---

### GETEVENT event trapping

The GETEVENT command can trap a mouse, key, message, or idle event and create a dynamic array with elements that represent the attributes of that event. The dynamic array created by GETEVENT contains all the elements that describe a valid event; these elements are discussed in detail in the section called “Valid events” later in this chapter. For mouse events, key events, and MENUKEY message events, the dynamic array created by GETEVENT also contains the elements described in Table 13-3.

Table 13-3 Additional elements returned by GETEVENT

Index	Value
CTRL	True if any <i>Ctrl</i> key is held down; False otherwise
ALT	True if any <i>Alt</i> key is held down; False otherwise
RIGHT SHIFT	True if the right <i>Shift</i> key is held down; False otherwise
LEFT CTRL	True if the left <i>Ctrl</i> key is held down; False otherwise
LEFT ALT	True if the left <i>Alt</i> key is held down; False otherwise
LEFT SHIFT	True if the left <i>Shift</i> key is held down; False otherwise
NUM LOCK	True if <i>Num Lock</i> is on; False otherwise
CAPS LOCK	True if <i>Caps Lock</i> is on; False otherwise
SCROLL LOCK	True if <i>Scroll Lock</i> is on; False otherwise

You can learn about GETEVENT by using the command to trap and display event attributes onscreen. The following example uses GETEVENT to trap key events and display their attributes on a canvas; the key combination *Ctrl-Q* quits the script.

```

ECHO NORMAL
WINDOW CREATE TO DisplayWindow      ; create a canvas to display
                                      ; GETEVENT key tags
WHILE True
  GETEVENT KEY "ALL" TO EventArray  ; creates a dynamic array for key events
  IF EventArray["KEYCODE"] = 17    ; if key Ctrl-0
    THEN QUIT                       ; then quit the script
    ELSE                             ; otherwise write key tags on the canvas
      CLEAR
      @0,0
      FOREACH Element IN EventArray
        ? Element, " holds the value ", EventArray[Element]
      ENDFOREACH
    ENDIF
  ENDWHILE

```

Notice how GETEVENT acts as a *filter* in this example. All events flow into this application, where the GETEVENT loop filters out the key events and lets the other events pass through to Paradox for normal processing. The key events are intercepted by GETEVENT and handled internally by the application—in this case, the key events are simply displayed on a canvas.

If you use your mouse while the script is running, you'll find that you can move or resize *DisplayWindow*—mouse events pass through to Paradox. You can even close *DisplayWindow*, although you will cause a run-time error when the script encounters the CLEAR statement. You can move, resize, and close this window because Paradox processes these events as it normally does. The GETEVENT command in the preceding example is only filtering *key* events.

---

### **Localizing events**

The LOCALIZEEVENT command provides an easy way to determine in which window a mouse event occurred when there is more than one window onscreen. LOCALIZEEVENT is only available for mouse events; it cannot be used with any other events or triggers.

Mouse events are represented in dynamic arrays with screen-relative coordinates for the ROW and COL tags. To easily determine the window-relative coordinates of a mouse event, use the LOCALIZEEVENT command after you use GETEVENT to trap a mouse event. LOCALIZEEVENT maps the screen-relative ROW and COL tags into window-relative coordinates.

LOCALIZEEVENT takes a dynamic array that represents a valid mouse event as an argument, converts the ROW and COL tags, and adds a WINDOW tag whose value represents the handle of the window where the event occurred.

For example, the following script creates a canvas window and solicits a mouse click within the window; after the mouse click, the canvas window displays the screen-relative coordinates and window-relative coordinates.

```

ECHO NORMAL
WINDOW CREATE TO MouseClickWindow
? "Click the mouse anywhere in this window."
?

GETEVENT MOUSE "DOWN" TO MouseArray
? "The screen-relative coordinates are:"
? "Row " + STRVAL(MouseArray["ROW"])
? "Column " + STRVAL(MouseArray["COL"])
?

LOCALIZEEVENT MouseArray ; LOCALIZEEVENT changes the
? "The window-relative coordinates are:" ; MouseArray dyanamic array
? "Row " + STRVAL(MouseArray["ROW"]) ; the ROW tag is different
? "Column " + STRVAL(MouseArray["COL"]) ; the COL tag is different
? ; the WINDOW tag is added
? "The window handle is " + STRVAL(MouseArray["WINDOW"])

```

You can also use the WINDOWAT() function to determine the handle of the topmost window in the z-order at a specified location. WINDOWAT() lets you specify the row and column coordinates for a location as follows:

```
RETURN WINDOWAT (1,5) ; returns window handle
```

In the above code fragment, the WINDOWAT() function returns the handle of the window that is at row 1, column 5.

You can combine WINDOWAT() with GETEVENT to conveniently determine the location of a specific event. For example, the following script uses GETEVENT to trap for mouse clicks and then uses WINDOWAT() to determine if the click occurred within a window:

```

WINDOW CREATE FLOATING TO CanvasWin ; create a window
GETEVENT MOUSE "DOWN" TO MouseArray ; trap mouse clicks
; evaluate location of mouse click with WINDOWAT()
IF WINDOWAT(MouseArray["ROW"], MouseArray["COL"]) <> 0
    THEN MESSAGE "You clicked in window ",
        WINDOWAT(MouseArray["ROW"], MouseArray["COL"])
    ELSE MESSAGE "You missed!"
ENDIF
SLEEP 2000

```

---

## WAIT event trapping

The WAIT command causes your application to pause while the user views or edits a table on the workspace. Every action the user makes generates one or more events. The WAIT command lets all events, except those specified in the *EventList*, pass through for processing by Paradox; a special procedure called the *WaitProc* is called for events trapped by the *EventList*. Chapter 16, "Using the WAIT command," contains a complete description of the WAIT command.

A WAIT interaction lets your script cooperate with Paradox and the user. You construct a WAIT statement so that your application waits for certain events and then responds to them. Paradox handles all of the other events that may take place while a user is editing or viewing a table.

You can wait while a user interacts with a multi-table form, a table, record, or field. `WAIT WORKSPACE` lets the user interact with a multi-table form or the entire workspace; `WAIT TABLE` allows the user to move through an entire table without ending the `WAIT`; `WAIT RECORD` confines the user to a single record in a table; and `WAIT FIELD` restricts the user to a single field. `WAIT` statements can be nested; you can place a `WAIT FIELD` inside a `WAIT RECORD`, and put the whole thing inside a `WAIT TABLE`.

During a `WAIT` interaction, all validity checks and image settings remain in effect. See “Validity checks” in Chapter 11 of the *User’s Guide*.

A `WAIT` statement can be constructed in one of two forms, in the form of `WAIT...UNTIL` or in the form which uses a `WAIT` procedure. `WAIT...UNTIL` allows your application to wait until the user presses specific keys. This form of the `WAIT` syntax was available in versions of Paradox prior to 4.0. The form of the `WAIT` syntax that uses the `WAIT` procedure is considerably more powerful because it allows you to trap for any type of event, including mouse events, and for triggers that indicate important interactions, such as `DEPARTFIELD` and `ARRIVEROW`. See “Using triggers” at the end of this chapter for complete information about triggers.

---

## **SHOWDIALOG event trapping**

The `SHOWDIALOG` command creates a dialog box onscreen to solicit information from the user. A `SHOWDIALOG` dialog box is *modal*; the user must cancel or accept the dialog box before interacting with anything outside the dialog box. Chapter 15, “Creating dialog boxes,” contains a complete description of the `SHOWDIALOG` command.

Every action that the user makes generates an event. If an event is meaningful in the context of the dialog box, it is interpreted by the `SHOWDIALOG` command. If the event is not meaningful in the context of the dialog box, it is eaten by the `SHOWDIALOG` command; it is not passed on to Paradox.

Like `WAIT`, `SHOWDIALOG` can use a procedure called for specific events. The `SHOWDIALOG` command lets all events except those specified in the *EventList* get interpreted or ignored by the dialog box; the *DialogProc* procedure is for events trapped by *EventList*.

The event or trigger that the dialog procedure is called for is still *pending* when the dialog procedure terminates. If you return the value `True` from the dialog procedure, the pending event is passed on to the dialog box and processed normally. If you return the value `False` from the dialog procedure, the pending event is *denied*; that is, it is not passed on to the dialog box and processed.

SHOWDIALOG, like WAIT, allows you to trap for any type of event, including mouse events, and for triggers that indicate important interaction. See "Using triggers" at the end of this chapter for complete information about triggers.

---

## Event blocking

The GETCHAR() function and the ACCEPT command let you prompt the user to respond by entering information from the keyboard. GETCHAR() and ACCEPT block all events except the specific key events they are expecting; no events are passed through to Paradox. GETCHAR() and ACCEPT differ from GETEVENT, WAIT, and SHOWDIALOG, which filter some events and let all others pass through to Paradox. GETCHAR() and ACCEPT ignore any mouse, message, or idle events.

---

## The GETCHAR() function

The simplest form of interaction with the user is the GETCHAR() function. GETCHAR() simply returns the keycode of the next character the user types or, if the user is typing ahead while PAL is doing something else, the next character in the keyboard input buffer.

Use GETCHAR() when you want to get only one or two characters from a user, as in a simple choice:

```
WINDOW CREATE TO DisplayWindow
ECHO NORMAL

TEXT
    1. Display the report on the screen
    2. Print the report on the printer
    3. Skip the report

    Please enter your choice or [Esc] to cancel:
ENDTEXT

WHILE True
    Choice = GETCHAR()                ; assign the user's next
                                      ; keystroke to Choice
    SWITCH
        CASE Choice = 27 : QUIT        ; if Esc
        CASE Choice = 49 : PLAY "DispRpt" ; if 1
        CASE Choice = 50 : PLAY "PrnRpt"  ; if 2
        CASE Choice = 51 : PLAY "Resume"   ; if 3
    OTHERWISE :
        MESSAGE "You must type a 1, 2, or 3"
        SLEEP 1000
        MESSAGE ""
    ENDSWITCH
ENDWHILE
```

As you'll see in Chapter 14, you could create a menu to do the same thing with one of the SHOW commands. In fact, the SHOW commands are often preferable because they also allow the user to interact with the mouse.

A useful way to keep a message onscreen until the user presses a character is to display the message, and then use `x = GETCHAR()` to

wait for and then throw away the next keypress. In this case, the displayed message should add the phrase **“Press any key to continue...”**.

---

### **The ACCEPT command**

The ACCEPT command is the input equivalent of the FORMAT() function—it accepts a single value from the user and makes sure it fits within a specified set of constraints. For example,

```
WINDOW CREATE @3,5 WIDTH 70 HEIGHT 6 TO InputWindow

SETMARGIN 1                                ; set offset = 1 for text
? "Today is ", FORMAT("D7", TODAY())       ; write text to the canvas
? "Enter future date (or press [Enter] to accept today) : "
ECHO NORMAL
ACCEPT "D"                                  ; get a date input
  MIN TODAY()                               ; restrict input between today
  MAX 1/1/99                                 ; and Jan 1 1999
  DEFAULT TODAY()                           ; default today's date
TO InputDate                                ; assign user input to InputDate
```

This example displays the current date and asks for a future one. The user can type a date or press *Enter* for the default (today), but dates before today or after January 1, 1999, are automatically rejected.

---

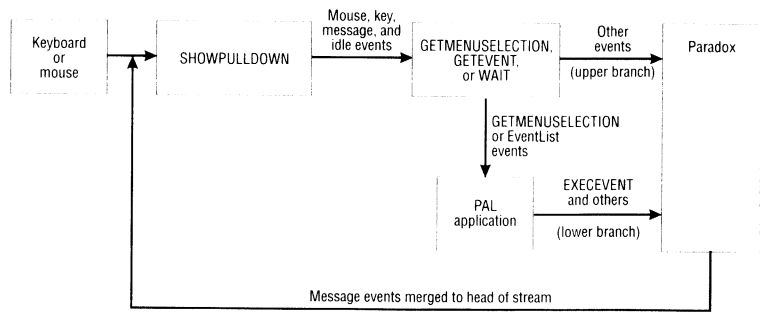
## **Typical application event processing**

A typical PAL application combines a SHOWPULLDOWN menu with a GETMENUSELECTION statement, a GETEVENT loop, or a WAIT interaction. Chapter 14, “Creating menus,” contains a complete description of the SHOWPULLDOWN command. The following section of this chapter gives you a brief overview of the type of event processing that frequently occurs within an application.

Whether you use SHOWPULLDOWN with GETMENUSELECTION, a GETEVENT loop, or a WAIT interaction, the event stream is similar. This event stream is shown in Figure 13-1.



Figure 13-1 Event stream in a typical application



## SHOWPULLDOWN and GETMENSELECTION

A menu created with SHOWPULLDOWN can be used with GETMENSELECTION when nothing is on the workspace to force a user to interact with the menu. At the beginning of an application, for example, you may want a user to make a choice from a menu that will give instructions to your application.

In this situation, the GETMENSELECTION/SHOWPULLDOWN combination forces the user to make a choice from the menu. GETMENSELECTION is used here to trap and discard all events except the MENSELECT and MENUKEY messages that arise from a SHOWPULLDOWN interaction. The event stream in this situation must flow through the lower path shown in Figure 13-1.

The use of SHOWPULLDOWN with a GETMENSELECTION is demonstrated in Chapter 24, "The Sample application," and Chapter 14, "Creating menus."

## SHOWPULLDOWN and WAIT

A SHOWPULLDOWN menu is frequently used in a WAIT interaction to let the user choose to close the WAIT table, open another table, print a report, access utilities, and so forth. Because the SHOWPULLDOWN menu is not modal, the user is free to interact with the WAIT table; the SHOWPULLDOWN menu is available when the user wants it.

Because the primary interaction is between the user and the WAIT table, many events will pass directly through your WAIT to Paradox for processing. This event stream is shown in the upper path of Figure 13-1. When the *WaitProc* is called for an event or trigger, your application takes over the processing. This event stream is shown in the lower path of Figure 13-1.

The use of SHOWPULLDOWN with a WAIT is also demonstrated in the Sample application discussed in Chapter 24.

---

## SHOWPULLDOWN and GETEVENT

A SHOWPULLDOWN menu is frequently combined with a GETEVENT loop to let the user interact with both the menu and windows on the desktop. The SHOWPULLDOWN menu in this situation is not modal; the user is free to interact with the workspace, but the SHOWPULLDOWN menu is available when the user wants it. Because the WAIT command provides more control with image windows, the available windows in this type of interaction are usually canvas windows.

Like the SHOWPULLDOWN/WAIT interaction, many events will pass directly through your GETEVENT loop to Paradox for processing. This event stream is shown in the upper path of Figure 13-1. When an EventList event occurs, your application takes over the processing. This event stream is shown in the lower path of Figure 13-1.

The use of SHOWPULLDOWN with a GETEVENT is demonstrated in Chapter 14, "Creating menus."

---

## Valid events

A *valid* event is represented by a dynamic array that can be correctly interpreted by Paradox as an event. The dynamic array that represents a valid event must contain the tags needed to completely identify the event; if other tags are present, they must not interfere with Paradox's ability to interpret the event.

The EXECEVENT command lets you execute a valid event that is defined in a dynamic array. You can use EXECEVENT to execute an event that you have trapped with GETEVENT. If you create a dynamic array with elements that completely and correctly represent a valid event, you can also use EXECEVENT to execute that event. See the description of EXECEVENT in the *PAL Reference* for an example of trapping a key event with GETEVENT, changing the description of the event, and using EXECEVENT to execute a different event.

---

## Valid mouse events

The dynamic array elements that represent a valid mouse event are described in Table 13-4.

Table 13-4 Elements of a valid mouse event

Index	Description
TYPE	MOUSE for a mouse event
ACTION	The specific mouse event that occurred; either DOWN, UP, MOVE, or AUTO
ROW	The row number on the screen
COL	The column number on the screen
BUTTONS	LEFT for the left mouse button, RIGHT for the right button, BOTH for both buttons (or the center button on a 3-button mouse), NONE if no button was pressed
DOUBLECLICK	True for a double-click; False otherwise

A valid mouse DOWN or AUTO event must specify a value other than NONE for the BUTTONS tag.

## Valid key events

The dynamic array elements that represent a valid key event are described in Table 13-5.

Table 13-5 Elements of a valid key event

Index	Description
TYPE	KEY for a key event
SCANCODE	Number that represents the code associated with the key by the system BIOS
KEYCODE	Number that represents the ASCII or IBM extended code

You do not need to specify both a SCANCODE and a KEYCODE to represent a key in a dynamic array for EXECEVENT. This information is redundant, and if the SCANCODE and KEYCODE contradict each other, the SCANCODE will have priority. The valid KEYCODE numbers are listed in Appendix G in the *PAL Reference*.

## Valid message events

The dynamic array elements that represent a valid message event are described in Table 13-6.

Table 13-6 Elements of a valid message event

Index	Description
TYPE	MESSAGE for a message event
MESSAGE	The specific message event that occurred; either CLOSE, MAXIMIZE, NEXT, MENUSELECT, or MENUKEY
MENUTAG	For MENUSELECT and MENUKEY, the TAG associated with a SHOWPULLDOWN menu item

Index	Description
SCANCODE	For MENUSELECT and MENUKEY, the number that represents the code associated with the key by the system BIOS
KEYCODE	For MENUSELECT and MENUKEY, the number that represents the ASCII or IBM extended code

Executing a MENUTAG or MENUKEY message event with EXECEVENT has no effect; this event is only meaningful to a PAL application, not to Paradox.

## Valid idle events

The dynamic array that represents a valid idle event needs to contain only a single tag, "TYPE", that is assigned the value "IDLE".

Executing an idle event with EXECEVENT has no noticeable effect; this event is only meaningful to a PAL application, not to Paradox. See the section called "Using the REPAINTDIALOG command" in Chapter 15, "Creating dialog boxes," for an example of using idle events.

## Using triggers

Strictly speaking, triggers are not events in the same sense as mouse, key, message, and idle. Triggers are an artificial category of events that are available only to the PAL SHOWDIALOG and WAIT commands. Triggers represent very specific interactions that occur during a WAIT or SHOWDIALOG command that are frequently useful for PAL applications to know about.

Like a message event, a Paradox trigger arises from a specific user action, independent of whether the action was caused by the keyboard or the mouse. The SHOWDIALOG and WAIT commands can trap triggers and call the *DialogProc* or *WaitProc* procedures.

Table 13-7 describes the different kinds of triggers that the SHOWDIALOG command can trap.

Table 13-7 SHOWDIALOG triggers

Trigger	Category	Caused by
UPDATE	Before	Attempting to assign the value expression of a control to its variable
DEPART	Before	Attempting to leave a control
SELECT	Before	Attempting to select a pick list item with <i>Space</i> or a double-click
ACCEPT	Before	Attempting to accept a dialog box

<b>Trigger</b>	<b>Category</b>	<b>Caused by</b>
CANCEL	Before	Attempting to cancel a dialog box
ARRIVE	After	Arriving on a new control
OPEN	After	Opening a dialog box but before it is shown
CLOSE	After	Closing a dialog box

As mentioned in Table 13-7, an UPDATE trigger occurs when you attempt to assign the value expression of a control element to its variable. However, the control element variable is bound in different situations for each control element. Table 13-8 describes the actions that result in an UPDATE trigger when different SHOWDIALOG control elements are active.

Table 13-8 SHOWDIALOG UPDATE triggers

<b>Control element</b>	<b>Action that causes UPDATE trigger</b>
Push button	Clicking the push button
Pick list	Moving within the pick list
Accept	Departing the accept control element
Slider	Moving the slider
Radio button	Selecting a radio button
Check box	Selecting or deselecting a check box

Table 13-9 describes the different kinds of triggers that the WAIT command can trap.

Table 13-9 WAIT triggers

<b>Trigger</b>	<b>Category</b>	<b>Caused by</b>
DEPARTFIELD	Before	Attempting to leave a field
DEPARTROW	Before	Attempting to leave a record
TOUCHRECORD	Before	Attempting to touch any record in Edit mode; attempting to touch a previously untouched record in Coedit mode
DEPARTPAGE	Before	Attempting to leave a form page
DEPARTTABLE	Before	Attempting to leave a table
POSTRECORD	Before	Attempting to post a record
IMAGERIGHTS	Before	Attempting to modify a table that was made read only with the IMAGERIGHTS READONLY command
PASSRIGHTS	Before	Attempting an action in a password protected field without sufficient rights

Trigger	Category	Caused by
REQUIREDVALUE	Before	Attempting to leave a required-value field without providing a value
VALCHECK	Before	Attempting an action that results in a validity check violation
DISPLAYONLY	Before	Attempting to modify a display-only field on an embedded detail
READONLY	Before	Attempting to change a table that was made read only at the the DOS or network level
ARRIVEPAGE	After	After arriving on a new form page
ARRIVEWINDOW	After	After arriving in a new window
ARRIVETABLE	After	After arriving in a new table
ARRIVEROW	After	After arriving in a new record
ARRIVEFIELD	After	After arriving in a new field

As you can see from Table 13-7 and 13-9, triggers are generated in situations that are convenient for PAL applications. For example, to find out when a user is trying to leave a field in a WAIT table, your application simply has to trap for a DEPARTFIELD trigger. Without triggers, your application would have to trap for every possible mouse and keyboard action that could result in the user leaving that field.

## Trigger categories

PAL divides triggers into two categories: *before* and *after*. A *before* trigger is issued when a user is *attempting* an action, before the action has taken place. An *after* trigger is issued after an action has taken place.

The distinction between these trigger categories is important. If your application traps for *before* triggers, you can *deny* a user action by intercepting the trigger and calling the appropriate procedure. *After* triggers are simply a notification that a user action has occurred. You deny a user action by returning False from the *DialogProc* procedure in SHOWDIALOG, and by returning either 1 or 2 from the *WaitProc* procedure in the WAIT. The trigger categories are also shown in Table 13-7 and Table 13-9. See Chapter 15, "Creating dialog boxes," for complete information about SHOWDIALOG and Chapter 16, "Using the WAIT command," for complete information about the WAIT.

## Trigger sequence

Within the SHOWDIALOG and the WAIT commands, triggers occur in specific sequences. In the SHOWDIALOG command, the sequence depends on the control that was active and the user action; in the WAIT command, the sequence depends on the location of the cursor, the type of WAIT (FIELD, RECORD, TABLE, or WORKSPACE) and the user action.

---

**WAIT triggers**

In the WAIT command, a trigger cycle is the complete set of procedure calls that arise from a single user action. For example, a single user action, such as leaving the last field in a record by pressing →, would result in the DEPARTFIELD, DEPARTROW, ARRIVEROW, and ARRIVEFIELD triggers issued in the sequence listed here. These triggers all occur within the same trigger cycle because they all arise from the same single action.

The triggers in any trigger cycle occur in a specific order. Because a single user action can result in a number of triggers, PAL provides an opportunity for the developer to step into a very specific part of a trigger cycle. Example 16-3 in Chapter 16 shows how to determine the specific sequence of triggers in a WAIT.

---

**SHOWDIALOG triggers**

Although there is no explicit cycle number available in the SHOWDIALOG command, SHOWDIALOG triggers are still issued in a specific sequence. For example, in a SHOWDIALOG dialog box with a pick list and an OK button, clicking the OK button when the pick list has focus results in the triggers DEPART, ARRIVE, UPDATE, ACCEPT, and CLOSE issued in the sequence listed here. Example 15-1 in Chapter 15 shows how to determine the specific sequence of triggers in a SHOWDIALOG dialog box.





# Creating menus

PAL lets you create different types of menus for different situations in your application. You can create both pop-up and pull-down menus with submenus. PAL menus let you take advantage of many of the features of Paradox.

PAL menus are useful for soliciting information from the user. In addition to these menus, PAL also lets you use dialog boxes to solicit responses from the user. See Chapter 15, "Creating dialog boxes," for information about using `SHOWDIALOG`, `SHOWARRAY`, `SHOWFILES`, and `SHOWTABLES` to create dialog boxes.

This chapter covers

- pop-up menus created with the `SHOWMENU` and `SHOWPOPUP` commands
- pull-down menus created with the `SHOWPULLDOWN` command

---

## Displaying a menu

You can use the `SHOWPULLDOWN`, `SHOWPOPUP`, and `SHOWMENU` commands to display a menu of choices to the user. The pull-down menus created with `SHOWPULLDOWN` are displayed at the top of the screen, with the main menu items on the top line and a description of the highlighted choice on the bottom line. The pop-up menus created with `SHOWMENU` are automatically placed in the center of the screen for you. The pop-up menus created with the `SHOWPOPUP` command can be placed anywhere onscreen.

In compatibility mode (and in versions of Paradox prior to 4.0), `SHOWMENU` creates ring menus at the top of the screen. `SHOWMENU` is available in standard mode, but `SHOWPULLDOWN` and `SHOWPOPUP` give you complete control over menu creation in standard mode.

The pull-down and pop-up menus created by PAL handle keypresses in exactly the same way as Paradox's built-in menus: Users can directly choose menu options by pressing the first letter of any option, using the direction keys to navigate through the menu, or using the mouse. You can add the UNTIL keyword to any of the PAL menus to handle specified keypresses that aren't automatically acted upon.

All pop-up menus created by PAL are *modal* menus. This means that the application stops running until the user chooses an item or presses *Esc* to clear the menu.

A menu created with the SHOWPULLDOWN command can be *modal* or *non-modal* menu. A non-modal menu is always *displayed* on the screen, but it is not always *active*. When a SHOWPULLDOWN menu is active, an application is suspended until the user completes the menu interaction; when a SHOWPULLDOWN menu is inactive, an application can continue running.

---

## SHOWMENU command

The SHOWMENU command creates a pop-up menu in the center of the desktop. SHOWMENU uses a comma-separated list of menu items (the *MenuList*) in the following form:

*Choice* : *Description*

To use the SHOWMENU command, supply a name for each choice to use on the menu (the *Choice*) and an explanation for each choice to use at the bottom of the screen (the *Description*). The *MenuSelection* variable that follows the TO keyword stores the *Choice* that the user selects. If the user presses *Esc* or clicks the mouse outside a SHOWMENU menu, the menu is removed and the value "Esc" is assigned to the TO variable. Following is the complete syntax of the SHOWMENU command:

### SHOWMENU

*MenuList*

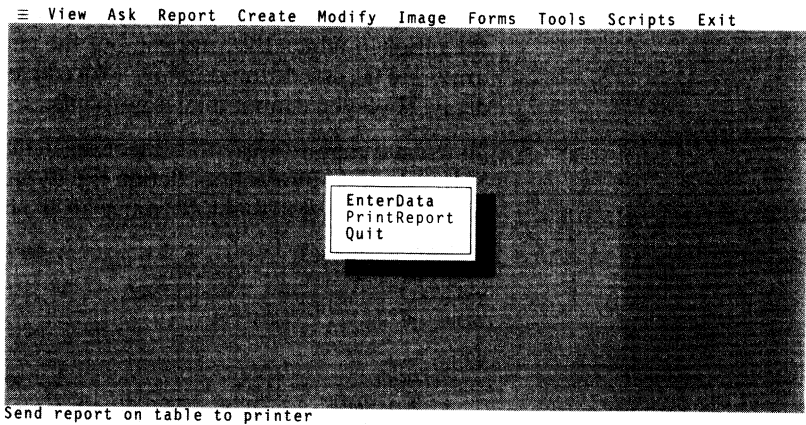
[ UNTIL *KeycodeList* [ KEYTO *KeyVar* ] ]

[ DEFAULT *Choice* ]

TO *MenuSelection*

You can use the optional DEFAULT keyword to specify a default menu choice. The default choice is the choice that is highlighted when the menu is first displayed; it is the choice that is selected if the user simply presses *Enter* when the menu is first displayed. If no default is specified, the first choice on the menu is the default.

Figure 14-1 SHOWMENU pop-up menu



For example, the following code creates a SHOWMENU pop-up menu with the default choice *PrintReport*, as shown in Figure 14-1:

```
SHOWMENU
  "EnterData" : "Enter data into table".
  "PrintReport" : "Send report on table to printer".
  "Quit" : "End this script and return to Paradox"
  DEFAULT "PrintReport"
TO MenuSelection
```

Use the optional UNTIL keyword to handle one or more keypresses other than the keypresses SHOWMENU automatically handles. For example, the following code fragment lets the user press *Ctrl-Q* (keycode 17) to remove the menu:

```
SHOWMENU
  "EnterData" : "Enter data into table".
  "PrintReport" : "Send report on table to printer".
  "Quit" : "End this script and return to Paradox"
  UNTIL 17
  KEYTO KeyVar
  DEFAULT "PrintReport"
TO MenuSelection
```

The user can cancel the menu by pressing a key on the UNTIL list or by pressing *Esc*; however, the script behaves differently in these two cases. If the user cancels by pressing *Esc*, the variable *MenuSelection* is assigned the value "Esc". If the user presses *Ctrl-Q*, the variable *KeyVar* is assigned the value 17. Canceling the menu in either manner sets *Retval* to False. When you use the UNTIL keyword, the KEYTO keyword and the *KeyVar* are required.

---

## Controlling the action

To control what happens after the user makes a choice from a menu, you usually use either a SWITCH command to play a script or a dynamic array to call a procedure. There is a significant performance

advantage to using a dynamic array to control the action when the SHOWMENU contains a large number of choices. The CASE statements within a SWITCH command are processed by Paradox in the order in which they appear, but the choices processed within a dynamic array are accessed immediately. Procedures executed from a dynamic array, however, cannot take arguments.

To use a dynamic array to control the action that results from a SHOWMENU interaction, define the dynamic array, issue the SHOWMENU statement, then use an IF statement, as follows:

```
DYNARRAY MenuTask[] ; define the dynamic array
MenuTask["EnterData"] = "EnterProc"
MenuTask["PrintReport"] = "PrintProc"
MenuTask["Quit"] = "QuitProc"

WHILE True
  SHOWMENU ; display the menu
  "EnterData" : "Enter data into table",
  "PrintReport" : "Send report on table to printer",
  "Quit" : "End this script and return to Paradox"
  TO MenuItem ; assign selection to variable

  IF ISASSIGNED(MenuTask[MenuItem]) ; if user picks an item that we
  ; have a procedure defined for
  THEN EXECPROC MenuTask[MenuItem] ; execute the procedure defined
  ; in the dynamic array
  ELSE BEEP ; otherwise alert user
  MESSAGE "Select an item from the menu"
  SLEEP 1500
  MESSAGE ""

  ENDF
ENDWHILE
```

To use the SWITCH command to control what happens when each choice is made, you could use the following:

```
WHILE True
  SHOWMENU ; display the menu
  "EnterData" : "Enter data into table",
  "PrintReport" : "Send report on table to printer",
  "Quit" : "End this script and return to Paradox"
  TO MenuItem ; assign selection to variable

  SWITCH ; evaluate selected item and take action
  CASE MenuItem = "EnterData" : PLAY "Enterdta"
  CASE MenuItem = "PrintReport" : PLAY "Prnreprt"
  CASE (MenuItem = "Quit") OR
  (MenuItem = "Esc") : QUIT
  ENDSWITCH
ENDWHILE
```

---

## SHOWPOPUP command

The SHOWPOPUP command creates a menu that looks similar to the ones created by SHOWMENU; however, SHOWPOPUP gives you

complete control over the attributes of the menu. SHOWPOPUP lets you specify the following:

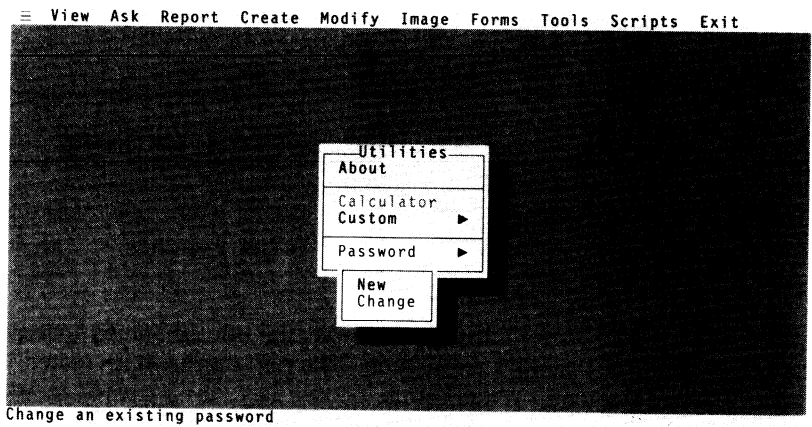
- title for the menu
- location of the menu
- up to 16 multiple levels for the menu
- disabled menu items
- separator bars between menu items

Like SHOWMENU, SHOWPOPUP uses a comma-separated *MenuList*, but the SHOWPOPUP menu items can be specified in any of the three following forms:

1. *Choice : Description : [ DISABLE ] Tag*
2. *Choice : Description : [ DISABLE ] [ Tag ]*  
**SUBMENU**  
*MenuList*  
**ENDSUBMENU**

### 3. SEPARATOR

Figure 14-2 SHOWPOPUP pop-up menu



Following is the complete syntax for SHOWPOPUP:

```
SHOWPOPUP Title { CENTERED | @Row, Column }
      MenuList
ENDMENU
[ UNTIL KeycodeList [ KEYTO KeyVar ]]
TO MenuSelection
```

Like SHOWMENU, the *Choice* is the name for each choice on the menu and the *Description* is the explanation displayed at the bottom of the screen. Unlike SHOWMENU, the *MenuSelection* variable that follows the TO keyword stores the *Tag* associated with the user's selection, not the *Choice*. Because the SHOWPOPUP *MenuSelection* variable stores the *Tag* and not the *Choice* itself, it is possible to have duplicate choices within the same menu. If the user presses *Esc* or clicks the mouse outside a SHOWPOPUP menu, the menu is removed and the value "Esc" is assigned to the *MenuSelection* variable.

The optional *Tag* on entries that introduce submenus is used to indicate the location of a keypress on the UNTIL *KeycodeList*. If the user cancels the menu by pressing a key on the UNTIL *KeycodeList*, the *Tag* associated with the item that was highlighted is assigned to the *MenuSelection* variable. If no *Tag* is used for that *Choice*, a blank string is assigned to the *MenuSelection* variable.

The following code creates the two-level pop-up menu shown in Figure 14-2; there are separators after *About* and before *Password*, and the *Calculator* choice on this menu is disabled:

```
SHOWPOPUP "Utilities" CENTERED
  "About"      : "About this application"      : "AboutTag",
  SEPARATOR,
  "Calculator" : "Display popup calculator"    : DISABLE "CalculatorTag",
  "Custom"    : "Customize colors or menus"   : "CustomTag"
  SUBMENU
    "Colors"   : "Customize screen colors"    : "ColorTag",
    "Menus"    : "Display custom menus"      : "MenuTag"
  ENDSUBMENU,
  SEPARATOR,
  "Password"  : "Create or change passwords"  : "PasswordTag"
  SUBMENU
    "New"      : "Create a new password"      : "NewTag",
    "Change"   : "Change an existing password": "ChangeTag"
  ENDSUBMENU
ENDMENU

UNTIL 17      ; Ctrl-Q to quit
KEYTO KeyVar
TO MenuVar
```

In this example, the title *Utilities* is specified within quotes as a string expression. This pop-up menu is created in the center of the screen, although it could have been created in any location by specifying @ *Row*, *Column* instead of CENTERED.

The UNTIL and KEYTO keywords are used the same way in SHOWPOPUP as they are in SHOWMENU. The action that occurs when the user makes a choice from the menu is controlled by a dynamic array or a SWITCH command, just as it is in SHOWMENU.

---

## SHOWPULLDOWN command

The SHOWPULLDOWN command creates a multi-level menu bar that looks similar to the menu bars that appear when you are running Paradox. As mentioned previously, SHOWPULLDOWN gives you the option of creating a modal or non-modal menu. Your application must be in a GETMENSELECTION loop to create a modal SHOWPULLDOWN menu and either a GETEVENT or WAIT loop to create a non-modal SHOWPULLDOWN menu.

When combined with the GETMENSELECTION command, SHOWPULLDOWN creates a modal menu, makes the menu active, and causes your PAL script to pause until the user selects an option. When combined with the GETEVENT or WAIT commands, SHOWPULLDOWN creates a non-modal menu but does not make the menu active; the SHOWPULLDOWN permits the PAL script to continue execution.

SHOWPULLDOWN uses the same comma-separated *MenuList* of menu items as SHOWPOPUP. Following is the complete syntax of the SHOWPULLDOWN command:

```
SHOWPULLDOWN  
    MenuList  
ENDMENU [ UNTIL KeycodeList ]
```

### *Modal SHOWPULLDOWN*

The modal SHOWPULLDOWN assigns values to the variables in the GETMENSELECTION command when the user interacts with the menu. Following is the complete syntax of the GETMENSELECTION command:

```
GETMENSELECTION  
    [ KEYTO KeyVar ]  
TO MenuSelection
```

Like SHOWPOPUP, the GETMENSELECTION *MenuSelection* variable that follows the TO keyword stores the *Tag* associated with the user's selection. If the user presses *Esc* or clicks the mouse outside the SHOWPULLDOWN menu, the menu is deactivated and the value "Esc" is assigned to the *MenuSelection* variable.

If the user cancels the menu by pressing a key on the SHOWPULLDOWN UNTIL *KeycodeList*, the *Tag* associated with the item that was highlighted is assigned to the GETMENSELECTION *MenuSelection* variable and the Keycode of the key that was pressed is assigned to the GETMENSELECTION KEYTO *KeyVar*. If no *Tag* is used for that *Choice*, a blank string is assigned to the *MenuSelection* variable. When you specify a SHOWPULLDOWN UNTIL *KeycodeList*, you must also specify a GETMENSELECTION KEYTO *KeyVar* if you want to receive the *Keycode* value.

Non-modal  
SHOWPULLDOWN

Unlike SHOWPOPUP, the non-modal SHOWPULLDOWN command uses a message event to indicate the action that a user takes. If the user selects a menu item, SHOWPULLDOWN generates a MENUSELECT message event and assigns the *Tag* associated with the menu item as the value of the MENUTAG tag. If the user presses one of the keys on the UNTIL list, SHOWPULLDOWN generates a MENUKEY message event; the SHOWPULLDOWN menu is deactivated, but it is not removed.

The dynamic array associated with the MENUKEY message contains an element whose tag is MENUTAG with a value that is the *Choice* that was active when the UNTIL key was pressed. The dynamic array associated with the MENUKEY message also contains additional elements to completely describe the keypress.

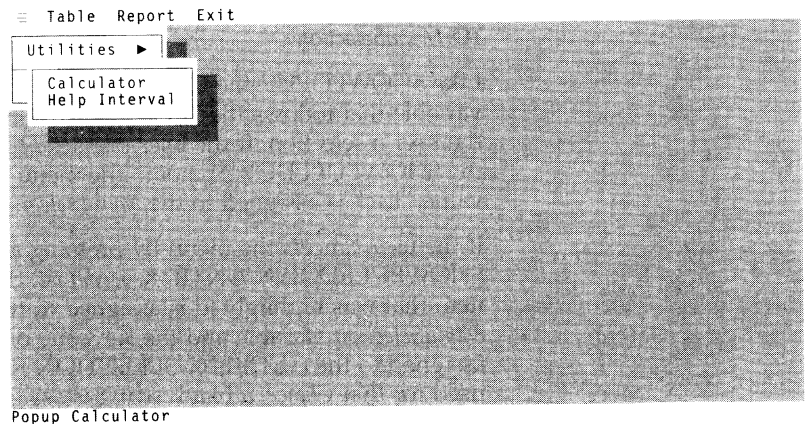
If you use the non-modal form of SHOWPULLDOWN, your application must trap for either MENUKEY or MENUSELECT messages; otherwise, the user will not be able to access the pull-down menu. Chapter 13, "Handling events," contains a complete description of the message events associated with SHOWPULLDOWN.

---

## SHOWPULLDOWN and GETMENSELECTION

A SHOWPULLDOWN menu is used with a GETMENSELECTION loop when you want the user to interact with the menu only and not with items on the desktop. At the beginning of an application, for example, you may want to let the user make a choice from a menu that will give instructions to your application when nothing is on the desktop.

Figure 14-3 SHOWPULLDOWN menu with an empty workspace



The following code creates the pull-down menu shown in Figure 14-3.



```

SHOWPULLDOWN
  "≡" : "Program Information." : "AltSpace"
  SUBMENU
    "Utilities" : "System Utilities" : "Utilities"
    SUBMENU
      "Calculator" : "Popup Calculator" : "Calculator",
      "Help Interval" : "Set AutoHelp Interval" : "Help Interval"
    ENDSUBMENU,
  SEPARATOR,
  "About" : "About this application." : "AltSpace/About"
  ENDSUBMENU,
  "Table" : "Work with the Invoice table." : "Table"
  SUBMENU
    "Modify" : "Modify Invoice table." : "Table/Modify",
    "Close" : "Close Invoice table." : "Table/Close"
  ENDSUBMENU,
  "Report" : "Report on Customer Table." : "Report"
  SUBMENU
    "All" : "Print report of all customers." :
      "Report/All",
    "West Coast" : "Print report of only West Coast customers" :
      "Report/West Coast"
  ENDSUBMENU,
  "Exit" : "Exit this application." : "Exit"
  SUBMENU
    "No" : "Do not exit application." : "Exit/No",
    "Yes" : "Exit this application." : "Exit/Yes"
  ENDSUBMENU
ENDMENU UNTIL 17 : Ctrl-Q to quit

WHILE True
  GETMENSELECTION KEYTO KeyVar TO MenuItemSelected
  IF Retval : if the user selects a menu item
    THEN QUITLOOP : quit the loop and take down the menu
  ENDF
  IF KeyVar = 17 : if the user presses Ctrl-Q
    THEN QUITLOOP : quit the loop and take down the menu
  ENDF
ENDWHILE

```

The code sample here places a `SHOWPULLDOWN` menu on the empty workspace, forcing the user to make a choice from the menu before continuing with the application. The `WHILE` loop that follows the `SHOWPULLDOWN` evaluates `Retval` to determine if the user selected a menu item and evaluates `KeyVar` to determine if the user pressed a key on the `UNTIL KeycodeList`. Because this script examines `Retval`, pressing `Esc` or clicking outside the menu will not cancel the `SHOWPULLDOWN` menu; the user must select a menu item or press `Ctrl-Q`.

If the user selects a menu item, `Retval` is set to `True` and the `SHOWPULLDOWN` menu is removed. If the user presses `Ctrl-Q`, `Retval` is set to `False` and the `SHOWPULLDOWN` menu is taken down. Because the `SHOWPULLDOWN` statement specifies an `UNTIL KeycodeList`, the `GETMENSELECTION` statement must specify a `KEYTO KeyVar` to receive the value of the `UNTIL Keycode`.

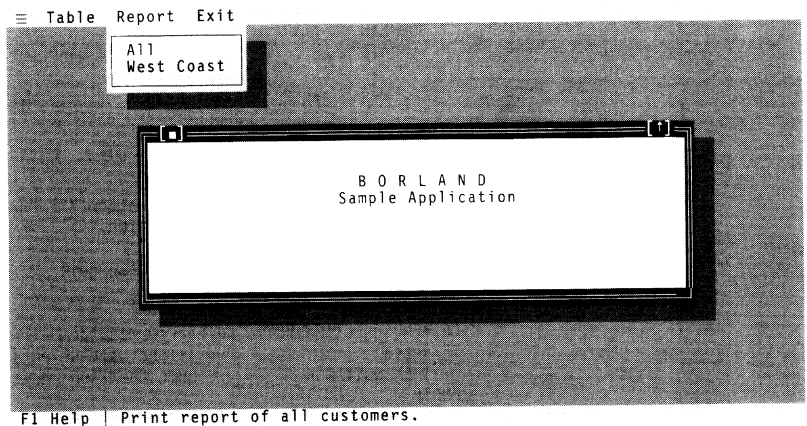
Based on the value of `MenuItemSelected` or `KeyVar`, a fully-developed application can specify the action that occurs after the menu

interaction is completed by using a dynamic array or a SWITCH command, just as with SHOWMENU.

## SHOWPULLDOWN and GETEVENT

A SHOWPULLDOWN menu is frequently combined with a GETEVENT loop to let the user interact with both the menu and windows on the desktop. Because the WAIT command provides more control with image windows, the available windows in this type of interaction are usually canvas windows.

Figure 14-4 SHOWPULLDOWN menu with a window on the workspace



The following code creates the SHOWPULLDOWN menu and displays the canvas window shown in Figure 14-4.

```
ECHO NORMAL
WINDOW CREATE @7,13 HEIGHT 11 WIDTH 56 to SplashScreen ; create a canvas window
@2,0 ?? FORMAT("W54,ac","B O R L A N D") ; and use it as a
@3,0 ?? FORMAT("W54,ac","Sample Application") ; splash screen

SHOWPULLDOWN
"=" : "Program Information" : "AltSpace"
SUBMENU
  "Utilities" : "System Utilities" : "Utilities"
  SUBMENU
    "Calculator" : "Popup Calculator" : "Calculator",
    "Help Interval" : "Set AutoHelp Interval" : "Help Interval"
  ENDSUBMENU,
  SEPARATOR,
  "About" : "About this application" : "AltSpace/About"
ENDSUBMENU,
"Table" : "Work with the Invoice table" : "Table"
SUBMENU
  "Modify" : "Modify Invoice table" : "Table/Modify",
  "Close" : "Close Invoice table" : "Table/Close"
ENDSUBMENU,
"Report" : "Report on Customer Table" : "Report"
SUBMENU
  "All" : "Print report of all customers" :
  "Report/All",
```

```

        "West Coast" : "Print report of only West Coast customers" :
                    "Report/West Coast"
    ENDSUBMENU.
    "Exit" : "Exit this application" : "Exit"
    SUBMENU
        "No" : "Do not exit application" : "Exit/No",
        "Yes" : "Exit this application" : "Exit/Yes"
    ENDSUBMENU
ENDMENU

WHILE True ; trap for MENUSELECT to see
    GETEVENT MESSAGE "MENUSELECT" TO EventInfo ; if the user selects a menu item
    MenuItemSelected = EventInfo["MENUTAG"] ; then assign the menu variable
    QUITLOOP ; and take down the menu --
ENDWHILE ; process other events normally

```

The code sample here places a `SHOWPULLDOWN` menu on the workspace with a canvas window that is used as a splash screen. The user is free to interact with either this canvas window or the pull-down menu; the `GETEVENT` loop following the `SHOWPULLDOWN` traps `MENUSELECT` message events that indicate the user has selected a menu item.

Because the `GETEVENT` loop filters only `MENUSELECT` message events, all other events are passed through to Paradox and processed normally; this is what enables the user to interact with the workspace. If the user selects a menu item, the value of the `MENUTAG` tag is assigned to the variable *MenuItemSelected*.

Based on the value of *MenuItemSelected*, the action that occurs when the user makes a choice from the menu is controlled by a dynamic array or a `SWITCH` command, just as it is in `SHOWMENU`.

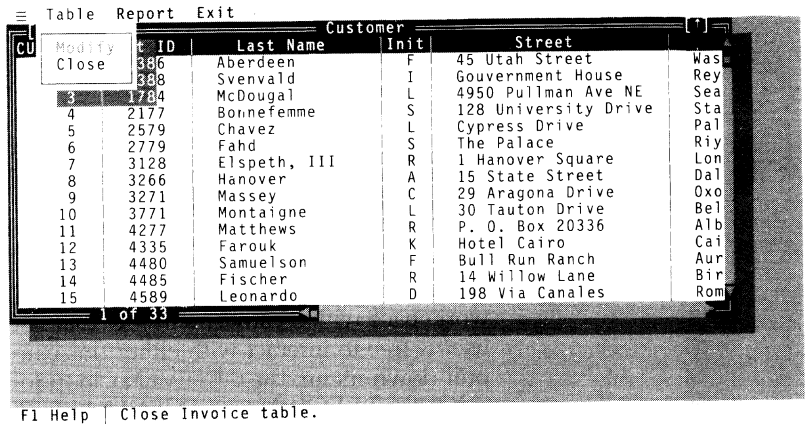
---

## **SHOWPULLDOWN and WAIT**

A `SHOWPULLDOWN` menu is frequently used in a `WAIT` interaction to let the user choose to close the `WAIT` table, open another table, print a report, access utilities, and so forth. Because the `SHOWPULLDOWN` menu is not modal, the user is free to interact with the `WAIT` table; the `SHOWPULLDOWN` menu is available when the user wants it.

Chapter 16, "Using the `WAIT` command," provides complete information about the `WAIT` command. The rest of this chapter explains how a `SHOWPULLDOWN` menu is typically used with the `WAIT` command.

Figure 14-5 SHOWPULLDOWN menu with a WAIT table



The following code creates the SHOWPULLDOWN menu and displays the WAIT WORKSPACE shown in Figure 14-5.

```

PROC WaitProc(EventType, EventRecord, CycleNumber)
    IF EventRecord["MENUTAG"] = "Exit/Yes"
        THEN DO IT!
            CLEARIMAGE
            ExitCode = 2 ; end the wait session
        ELSE MESSAGE EventRecord["MENUTAG"], " not installed"
            SLEEP 1500
            MESSAGE ""
            ExitCode = 0 ; process event, don't break
            ; wait session
    ENDIF
    RETURN ExitCode
ENDPROC

SHOWPULLDOWN
    "=" : "Program Information" : "AltSpace"
    SUBMENU
        "Utilities" : "System Utilities" : "Utilities"
        SUBMENU
            "Calculator" : "Popup Calculator" : "Calculator",
            "Help Interval" : "Set AutoHelp Interval" : "Help Interval"
        ENDSUBMENU,
        SEPARATOR,
        "About" : "About this application" : "AltSpace/About"
    ENDSUBMENU,
    "Table" : "Work with the Invoice table" : "Table"
    SUBMENU
        "Modify" : "Modify Invoice table" : "Table/Modify",
        "Close" : "Close Invoice table" : "Table/Close"
    ENDSUBMENU,
    "Report" : "Report on Customer Table" : "Report"
    SUBMENU
        "All" : "Print report of all customers" :
            "Report/All",
        "West Coast" : "Print report of only West Coast customers" :
            "Report/West Coast"
    ENDSUBMENU,

```

```

"Exit" : "Exit this application" : "Exit"
SUBMENU
  "No" : "Do not exit application" : "Exit/No",
  "Yes" : "Exit this application" : "Exit/Yes"
ENDSUBMENU
ENDMENU

COEDIT "Customer"

PROMPT "[F10] Menu"
WAIT WORKSPACE : let user interact with workspace
  PROC "WaitProc" : call procedure when user selects
  MESSAGE "MENUSELECT" : an item from the menu
ENDWAIT
CLEARPULLDOWN

```

---

## Disabling and enabling menu items

If you want to create a `SHOWPULLDOWN` menu with items that are available only in certain contexts, you can disable the items in inappropriate contexts. The optional `DISABLE` keyword lets you disable menu items at the time of creation. By default, menu items are created enabled. You can also enable and disable menu items after the menu is created with the `MENUENABLE` and `MENUDISABLE` commands.

Use `MENUENABLE` and `MENUDISABLE` followed by the *Tag* of the menu item that you want to enable or disable. For example,

```
MENUENABLE "Table/Modify"
```

would enable the menu item `Table|Modify` in the above code listing.

`MENUENABLE` and `MENUDISABLE` are typically used within the `WAIT` procedure to make the state of the pull-down menu consistent with the options that are available in the application at a certain time. For example, the menu shown in Figure 14-5 displays the choices `Table|Modify` and `Table|Close`. In Figure 14-5, `Table|Modify` is disabled because the workspace is already in `CoEdit` mode for the `WAIT` table. If this application began by displaying a pull-down menu on an empty workspace, `Table|Close` would be disabled because no table would be on the workspace.

If the user chooses `Table|Close` from the pull-down menu shown in Figure 14-5, you could use the `WAIT` procedure to change the state of the menu and to close the table.

The following code shows the WAIT procedure being called for a message MENUSELECT event; the WAIT procedure evaluates the event to determine if the user selected the menu item Table|Close:

```
PROC WaitProc (EventType, EventRecord, CycleNumber)
  IF EventRecord["TYPE"] = "MESSAGE"
    AND EventRecord["MESSAGE"] = "MENUSELECT"
    AND EventRecord["MENUTAG"] = "Table/Close"
  THEN WINDOW CLOSE           ; close the table
    MENUISABLE "Table/Close" ; disable Table|Close
    MENUENABLE "Table/Modify" ; enable Table|Modify
  ENDIF

...

ENDPROC
```

---

## Removing a pull-down menu

The pop-up menus created by PAL are removed when the user selects a menu item, presses *Esc*, or presses a key on the UNTIL list. The menus created with SHOWPULLDOWN, however, remain visible unless you do one of the following:

- explicitly close them by issuing the CLEARPULLDOWN command
- issue another SHOWPULLDOWN command
- change to compatible mode

You cannot “stack” SHOWPULLDOWN menus by issuing a SHOWPULLDOWN command while another is in effect. If a SHOWPULLDOWN command is issued while a previous SHOWPULLDOWN menu is visible, the first menu is removed.

You can create the effect of nested SHOWPULLDOWN menus by placing each SHOWPULLDOWN in a procedure and making the procedures call each other. For example, one procedure could make an application’s main menu visible, then call a second procedure to place a SHOWPULLDOWN menu for an editor session, which calls the first procedure when it is finished and displays the main menu again.

# Creating dialog boxes

PAL lets you create different types of dialog boxes for different situations in your application. The `SHOWARRAY`, `SHOWFILES`, and `SHOWTABLES` commands let you quickly create a dialog box that presents a pick list of array elements, files, or tables to the user. For more sophisticated dialog boxes, the `SHOWDIALOG` command lets you specify the exact set of characteristics and control elements you want to use. The `SHOWDIALOG` command also gives you access to an *EventList* for event-driven control.

Dialog boxes are useful for soliciting information from the user. In addition to dialog boxes, PAL also lets you use pop-up and pull-down menus to solicit responses from the user. See Chapter 14, “Creating menus,” for information about using `SHOWPULLDOWN`, `SHOWMENU`, and `SHOWPOPUP` to create menus.

This chapter covers:

- ❑ creating simple dialog boxes with `SHOWFILES`, `SHOWTABLES`, and `SHOWARRAY`
- ❑ creating complex dialog boxes with `SHOWDIALOG`
- ❑ specifying background canvas elements and control elements for dialog boxes
- ❑ using the dialog procedure to respond to user interactions
- ❑ controlling the behavior of a dialog box during user interactions with the `ACCEPTDIALOG`, `CANCELDIALOG`, `NEWDIALOGSPEC`, `REFRESHCONTROL`, `REFRESHDIALOG`, `REPAINTDIALOG`, `RESYNCCONTROL`, `RESYNCDIALOG`, and `SELECTCONTROL` commands and the `CONTROLVALUE()` function

---

## Displaying a dialog box

You can use the SHOWFILES, SHOWTABLES, SHOWARRAY, or SHOWDIALOG commands to display a dialog box filled with different types of choices for the user to make. All dialog boxes created by PAL are *modal* dialog boxes. This means that the application stops running until the dialog box is either accepted or cancelled.

In compatible mode (and in versions of Paradox prior to 4.0), SHOWFILES, SHOWTABLES, and SHOWARRAY create ring menus at the top of the screen. SHOWFILES, SHOWTABLES, and SHOWARRAY create dialog boxes in standard mode; however, SHOWDIALOG gives you complete control over dialog box creation in standard mode. SHOWDIALOG is available in standard mode only.

---

## Creating simple dialog boxes

SHOWARRAY, SHOWFILES, and SHOWTABLES dialog boxes automatically have the following attributes when they are created:

- dialog box is centered onscreen
- pick list is centered in dialog box
- first item in the pick list is selected
- OK and Cancel push buttons are centered at bottom of dialog box
- OK push button is the default (the push button that is pressed if the *Enter* key is pressed)

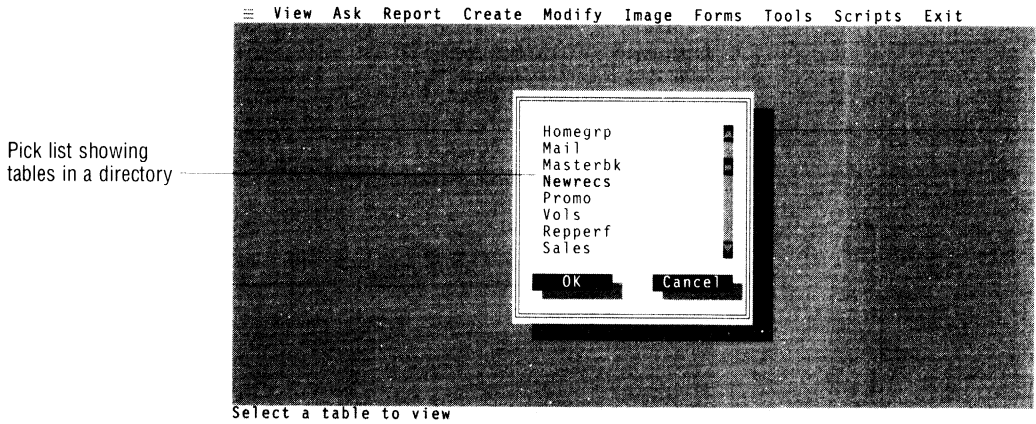
---

### SHOWFILES and SHOWTABLES dialog boxes

The SHOWFILES and SHOWTABLES commands let you quickly create dialog boxes that display a pick list of files or tables, respectively. Figure 15-1 shows a typical SHOWFILES or SHOWTABLES dialog box.



Figure 15-1 Typical SHOWFILES or SHOWTABLES dialog box



Select a table to view

To create a dialog box with the SHOWTABLES command, specify a path for the tables as a string, a prompt as a string, and a TO variable to accept the table name using the following syntax:

```
SHOWTABLES DOSPath Prompt
  [ UNTIL KeycodeList [ KEYTO KeyVar ] ]
TO SelectionVar
```

For example, the following code uses SHOWTABLES to create the dialog box shown in Figure 15-1; this example also lets the user select a table from the pick list to view:

```
DosPath = "C:\PDOX40\SAMPLE\"           ; specify the path
SHOWTABLES DosPath                       ; display tables in a dialog box
  "Select a table to view"                ; display a prompt onscreen
TO TableVar                              ; assign table name to variable

IF TableVar <> "Esc"                     ; if the user does not cancel
  THEN VIEW DosPath + TableVar           ; then view the table
ENDIF
```

You can create a dialog box that displays a pick list of files with the SHOWFILES command and similar syntax, shown below:

```
SHOWFILES [ NOEXT ] DOSPath Prompt
  [ UNTIL KeycodeList [ KEYTO KeyVar ] ]
TO SelectionVar
```

The SHOWFILES command lets you specify a file-name filter with the DOS wildcards \* and ?. If you do not specify a file-name filter with the path, the SHOWFILES dialog box displays all files in the directory. If you specify the optional NOEXT keyword, the pick list will not display the file-name extensions.

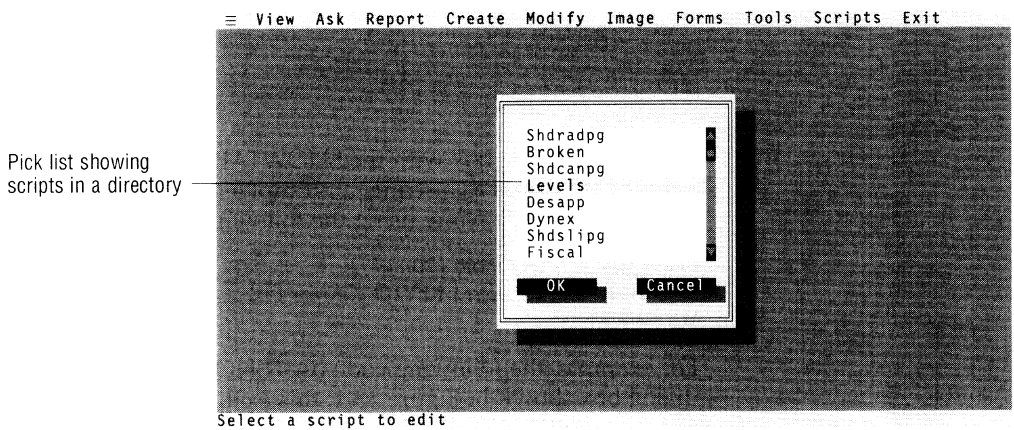
For example, the following code creates the dialog box shown in Figure 15-2; this dialog box lists all the scripts in a directory (suppressing the .SC extension) and lets the user select a script to edit:

```

PathName = "C:\\PDOX40\\SAMPLE\\" ; specify the path
SHOWFILES NOEXT PathName + "*.SC" ; display scripts in a dialog box
  "Select a script to edit" ; display a prompt onscreen
TO FileVar ; assign script name to variable
IF FileVar <> "Esc" ; if the user does not cancel
  THEN ; then edit the script
    EDITOR OPEN PathName + FileVar + ".SC"
ENDIF

```

Figure 15-2 SHOWFILES dialog box with pick list of scripts



## SHOWARRAY dialog boxes

The SHOWARRAY command lets you quickly create dialog boxes that display a pick list of array elements. SHOWARRAY creates a dialog box that looks similar to the ones created by SHOWFILES and SHOWTABLES, but the pick list items and prompts that are used by SHOWARRAY are stored in two fixed arrays that you specify. Following is the complete syntax of the SHOWARRAY command:

```

SHOWARRAY ChoiceArray DescriptionArray
  [ UNTIL KeycodeList [ KEYTO KeyVar ] ]
  [ DEFAULT Choice ]
TO SelectionVar

```

To use SHOWARRAY, create the two fixed arrays that are needed and then execute the command. For example, the following code creates a SHOWARRAY dialog box:

```

; define the array for pick list choices
ARRAY Choice[3]
  Choice[1] = "EnterData"
  Choice[2] = "PrintReport"
  Choice[3] = "Quit"

```

```

; define the array for prompt descriptions
ARRAY Description[3]
Description[1] = "Enter data into table"
Description[2] = "Send report on table to printer"
Description[3] = "End this script and return to Paradox"

; create the dialog box
SHOWARRAY Choice Description
UNTIL 17
KEYTO KeyVar
DEFAULT "PrintReport"
TO SelectionVar

```

The `DEFAULT` keyword lets you specify an item that is selected by default when the dialog box is created. The `UNTIL` and `KEYTO` keywords are used the same way in `SHOWARRAY` as they are in `SHOWFILES` and `SHOWTABLES`. The *SelectionVar* variable stores the value of the element in the *ChoiceArray* that represents the choice the user selects.

The previous example creates a `SHOWARRAY` dialog box with the default selection *PrintReport*. You can create a generic `SHOWARRAY` statement by using a variable to specify the default selection instead of a string. For example, the following code could replace the `SHOWARRAY` statement above:

```

; specify a dialog box default
DefaultChoice = 2

; create the dialog box
SHOWARRAY Choice Description
UNTIL 17
KEYTO KeyVar
DEFAULT Choice[DefaultChoice]
TO SelectionVar

```

---

## Creating complex dialog boxes

The `SHOWDIALOG` command creates a dialog box that can look similar to the ones created by `SHOWFILES` and `SHOWTABLES`; however, `SHOWDIALOG` gives you complete control over the attributes of the dialog box. `SHOWDIALOG` lets you specify the following:

- title for the dialog box
- size and location of the dialog box
- canvas elements, such as frames, colors, and text, to be placed within the dialog box
- control elements, including pick lists, radio buttons, check boxes, and others, in all combinations

- an optional dialog procedure that is called for specified events that occur within the dialog box

Following is the complete syntax of the SHOWDIALOG command:

```
SHOWDIALOG TitleExpression  
[ PROC DialogProc EventList ]  
  @ Row, Column HEIGHT Number WIDTH Number  
  CanvasElements and ControlElements  
ENDDIALOG
```

The SHOWDIALOG *TitleExpression* is a string expression that appears centered in the top of the dialog box frame as a title for the dialog box. When you create a dialog box, you specify its location with the @ *Row, Column* statement; you also specify the width and the height with the WIDTH and HEIGHT keywords, respectively.

The SHOWDIALOG *CanvasElements* are the static background elements such as frames, colors, and text. The *ControlElements* are the elements that the user can interact with, such as pick lists, radio buttons, and check boxes. You must specify at least one *ControlElement* (usually a push button) to create a valid dialog box. The *CanvasElements* and *ControlElements* can be specified in any order; the *CanvasElements* do not have to precede the *ControlElements*. The *EventList* is a comma-separated list of events that the optional procedure *DialogProc* is called for.

You must provide a way for the user to exit a dialog box. A valid dialog box can contain either an OK button, a Cancel button, or a dialog procedure to let the user exit.

---

## Using control elements in a dialog box

The SHOWDIALOG dialog box lets you specify any of the following control elements:

- push buttons
- pick lists
- radio buttons
- check boxes
- type-in boxes
- sliders

As mentioned previously, you must specify at least one control element to create a valid dialog box; usually that element is a push button. The order in which you specify control elements within the SHOWDIALOG command is the order in which these elements receive focus when the user presses the *Tab* key to move through the

dialog box. The first control element that you specify is the element that has focus when the dialog box is opened.

Interactions with a dialog box always set *Retval*. If a dialog box is accepted, *Retval* is set to True; if a dialog box is cancelled, *Retval* is set to False.

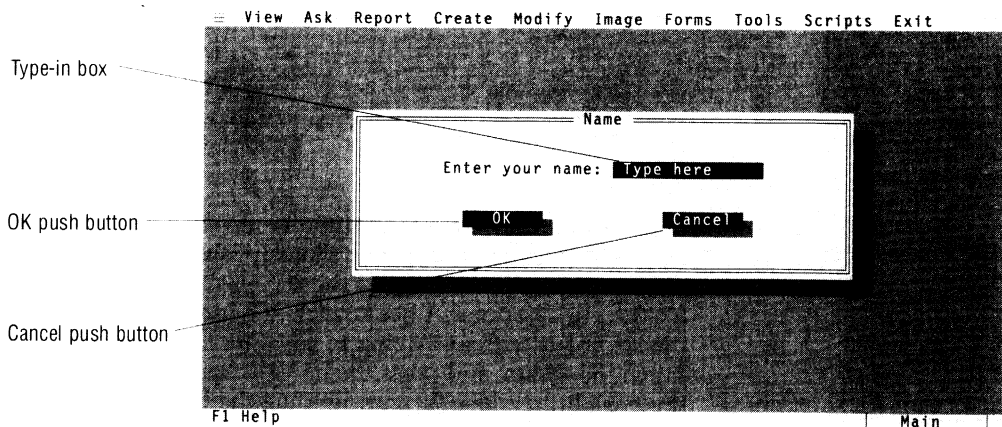
You do not need to specify *CanvasElements* or a *DialogProc* to create a valid dialog box, although you will frequently want to use one or both in your dialog boxes. To simplify the examples for the individual control elements, we do not specify *CanvasElements* or a *DialogProc*; later in this chapter, however, there are examples that combine control elements, canvas elements, and the dialog procedure.

---

### Push buttons

Push buttons are the most common type of control element in a dialog box. Every dialog box must provide a way for a user to exit; by convention, the choice for exiting is usually displayed as a pair of push buttons labeled OK and Cancel. The OK push button is used to accept selections the user makes within a dialog box and pass them on to the application; the Cancel push button is used to cancel the dialog box and ignore any selections made within the dialog box by the user. Figure 15-3 shows a typical SHOWDIALOG dialog box with OK and Cancel push buttons.

Figure 15-3 SHOWDIALOG with push buttons



In a SHOWDIALOG dialog box, the PUSHBUTTON statement lets you create a push button. The complete syntax of the PUSHBUTTON statement follows:

```
PUSHBUTTON  
  @Row, Column  
  WIDTH Number
```

```

Label
[ OK | CANCEL ]
[ DEFAULT ]
VALUE Expression
TAG Expression
TO VarName

```

When you create a push button, you specify its location with the *@ Row, Column* statement in coordinates that are relative to the inside of the dialog box; the upper left corner within the frame of the dialog box is the 0,0 coordinate. You also specify the width of the push button with the WIDTH keyword and assign a string expression as the *Label* that identifies the push button to the user. The expression that follows the VALUE keyword is assigned to the control element variable *VarName* if the push button is pressed and the dialog box is accepted; the expression that follows the TAG keyword identifies the current control element. The following code creates a simple dialog box with push buttons:

```

SHOWDIALOG "Dialog Box with Push Buttons"
@7,17 HEIGHT 9 WIDTH 44           ; location and dimension of box

PUSHBUTTON @5,8 WIDTH 10          ; create a push button
"~O~K"                             ; label it OK with O as hot key
OK                                  ; specify OK as type of action
VALUE "Accept"                     ; value of button when pressed
TAG "Yes"                           ; name passed to procedure as TagValue
TO ButtonValue                       ; variable assigned with VALUE value

PUSHBUTTON @5,23 WIDTH 10         ; create a push button
"~C~ancel"                          ; label it Cancel with C as hot key
CANCEL                              ; specify CANCEL as type of action
DEFAULT                             ; this button is selected by default
VALUE "Cancel"                     ; value of button when pressed
TAG "No"                             ; name passed to procedure as TagValue
TO ButtonValue                       ; variable assigned with VALUE value

ENDDIALOG

```

In this example, the Cancel button is specified as the default push button; this is the push button that is pushed if the user presses the *Enter* key. The "O" in OK and the "C" in Cancel are used as hot keys for the push buttons. A hot key lets the user press a push button from the keyboard without having to navigate to it. To use a hot key, hold down the *Alt* key and press the hot key. You can specify a hot key in the push button label by surrounding the hot key with tilde (~) characters.

---

### **Pick lists**

A pick list is a scrolling list box that is used to present a list of choices for the user to select from. PAL lets you make five types of pick lists in a SHOWDIALOG dialog box. You can create a pick list of files in a directory, tables in a directory, values of fixed array elements, values of dynamic array elements, or tags of dynamic array elements.

PICKFILE and PICKTABLE create a pick list of files and tables, respectively. PICKARRAY, PICKDYNARRAY, and PICKDYNARRAYINDEX create pick lists by taking advantage of the power and flexibility of fixed and dynamic arrays. The PICKFILE and PICKTABLE pick list elements provide functionality similar to the SHOWFILES and SHOWTABLES commands, but allow you the flexibility of working with a SHOWDIALOG dialog box.

---

**PICKFILE control element**

The PICKFILE control element lets you create a pick list with the dimensions and location that you specify. The complete syntax of the PICKFILE statement follows:

**PICKFILE**

*@Row,Column*  
**HEIGHT** *Number*  
**WIDTH** *Number*  
**[ COLUMNS** *Number* **]**  
*DOSPath*  
**[ NOEXT ]**  
**TAG** *Expression*  
**TO** *VarName*

When you create a pick list with PICKFILE, you specify its location with the *@Row, Column* statement in coordinates that are relative to the upper left corner of the dialog box. Use the HEIGHT and WIDTH keywords to specify the height and width of the pick list, respectively. You can optionally specify the number of columns to be used in the pick list with the COLUMNS keyword; the default is one column. The optional NOEXT keyword lets you display file names without their extensions.

*DOSPath* is a string expression that specifies the path for the files. You can specify a file-name filter with the DOS wildcards \* and ?. If you do not specify a file-name filter with the path, the pick list displays all files in the directory.

The pick list item that has focus is assigned to the control element variable *VarName* if the dialog box is accepted. The expression that follows the TAG keyword identifies the current control element.

You can specify a pick list item as the default selection when the dialog box is opened. To specify a default, assign the PICKFILE *VarName* a value before the SHOWDIALOG statement is issued. If the value you assign is the name of a file on the *DOSPath*, the file will appear selected when the dialog box is opened.

The following code creates a dialog box with two push buttons and a pick list, as shown in Figure 15-4:

```
FileDir = "C:\\PDOX40\\SAMPLE\\*.SC" ; define variable to represent directory  
; where files are located and filter to
```

```

                                : display scripts only
SHOWDIALOG "List of Scripts"    : begin dialog box definition
@5,23 HEIGHT 14 WIDTH 31

PICKFILE @1,3 HEIGHT 8 WIDTH 22 : define a PICKFILE element
  COLUMNS 2                    : display files in two columns
  FileDir                      : directory where files exist
  NOEXT                        : do not display file-name extensions
  TAG "ChosenFile"             : name passed to procedure as TagValue
  TO FileChoice                : variable assigned with tag value

PUSHBUTTON @10,2 WIDTH 10      : define OK push button
  "~O~K"
  OK
  DEFAULT
  VALUE "Accept"
  TAG "Yes"
  TO ButtonValue

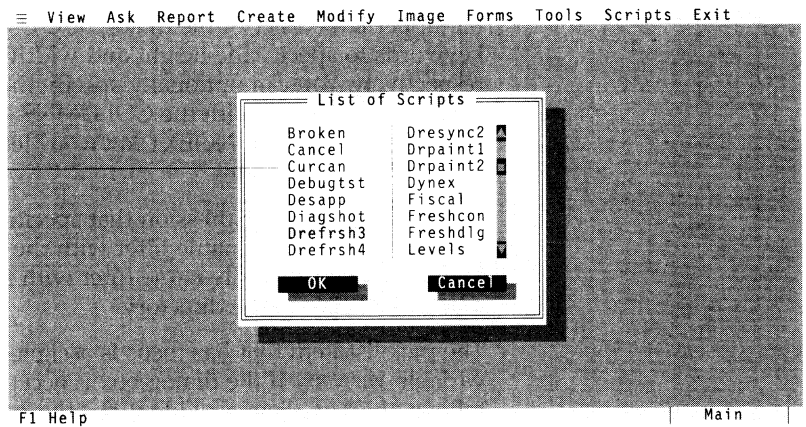
PUSHBUTTON @10,17 WIDTH 10     : define Cancel push button
  "~C~ancel"
  CANCEL
  VALUE "Cancel"
  TAG "No"
  TO ButtonValue

ENDDIALOG

```

Figure 15-4 Two-column pick list of scripts

Two-column pick list showing scripts without filename extensions



Notice how the above example uses the variable *FileDir* to specify the path where the scripts are located. Using a variable in this manner helps you keep the code for creating PICKFILE elements generic. If you frequently create 2-column pick lists, for example, you could always use the same PICKFILE statement and simply redefine the *FileDir* variable. Using the variable *FileDir* also provides the opportunity to dynamically redefine the pick list from the dialog procedure, as shown later in this chapter.



---

**PICKTABLE control element**

The PICKTABLE control element lets you create a pick list with the dimensions and location that you specify. The complete syntax of the PICKTABLE statement follows:

```
PICKTABLE  
  @ Row,Column  
  HEIGHT Number  
  WIDTH Number  
  [ COLUMNS Number ]  
  DOSPath  
  TAG Expression  
  TO VarName
```

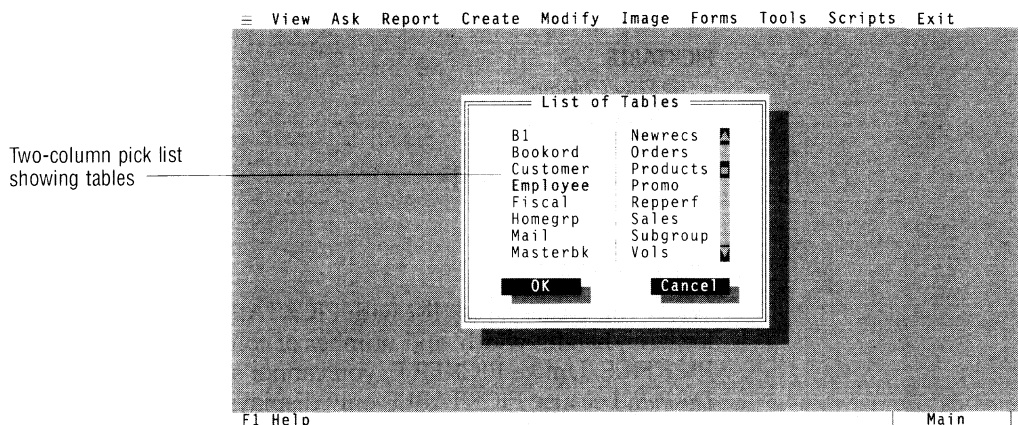
When you create a pick list with PICKTABLE, you specify its location, height, width, and number of columns just as you do for PICKFILE. Unlike PICKFILE, you cannot specify a file-name filter for *DosPath* because PICKTABLE only displays tables. The TAG *Expression* and TO *VarName* statements are handled by PICKTABLE the same way they are handled by PICKFILE.

You can specify a pick list item as the default selection when the dialog box is opened. To specify a default, assign the PICKTABLE *VarName* a value before the SHOWDIALOG statement is issued. If the value you assign is the name of a file on the *DOSPath*, the file will appear selected when the dialog box is opened.

The following code creates a dialog box with two push buttons and a pick list of tables, as shown in Figure 15-5:

```
TableDir = "C:\PDOX40\SAMPLE\" ; define variable to represent directory  
; where tables are located  
SHOWDIALOG "List of Tables" ; begin dialog box definition  
  @5,23 HEIGHT 14 WIDTH 31  
  
  PICKTABLE @1,3 HEIGHT 8 WIDTH 22 ; define a PICKTABLE element  
    COLUMNS 2 ; display tables in two columns  
    TableDir ; directory where tables exist  
    TAG "ChosenFile" ; name passed to procedure as TagValue  
    TO FileChoice ; variable assigned with tag value  
  
  PUSHBUTTON @10,2 WIDTH 10 ; define OK push button  
    "~O~K"  
    OK  
    DEFAULT  
    VALUE "Accept"  
    TAG "Yes"  
    TO ButtonValue  
  
  PUSHBUTTON @10,17 WIDTH 10 ; define Cancel push button  
    "~C~ancel"  
    CANCEL  
    VALUE "Cancel"  
    TAG "No"  
    TO ButtonValue  
  
ENDDIALOG
```

Figure 15-5 Two-column pick list of tables



**PICKARRAY control element**

The PICKARRAY control element lets you create a pick list with the dimensions and location that you specify. The complete syntax of the PICKARRAY statement follows:

**PICKARRAY**  
@*Row, Column*  
**HEIGHT** *Number*  
**WIDTH** *Number*  
[ **COLUMNS** *Number* ]  
*ArrayName*  
**TAG** *Expression*  
**TO** *VarName*

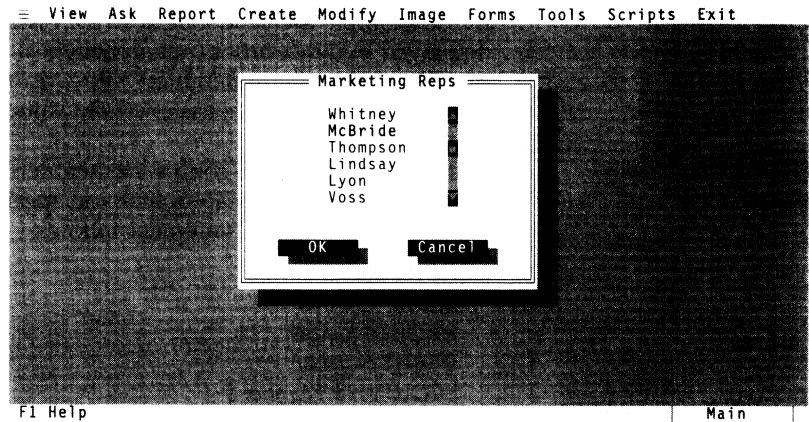
When you create a pick list with PICKARRAY, you specify its location, height, width, and number of columns just as you do for PICKFILE. The TAG *Expression* statement is handled by PICKARRAY the same way it is handled by PICKFILE.

The TO *VarName* statement is treated differently in PICKARRAY than it is in PICKFILE. The values of fixed array elements in *ArrayName* appear as items in the pick list. When a pick list item is selected, the index of the fixed array element is assigned to the PICKARRAY variable *VarName*. For example, if *DateArray[1]=7/17/56* is an element in a fixed array, *7/17/56* appears as an item in the pick list. If this item is selected, *VarName* is assigned the value *1* when the dialog is accepted.

You can specify a pick list item as the default selection when the dialog box is opened. To specify a default, assign the PICKARRAY *VarName* a value before the SHOWDIALOG statement is issued. If the value you assign is the index of an *ArrayName* element, that item will appear selected when the dialog box is opened.

The values in a PICKARRAY pick list are displayed in the index order. The following code creates a dialog box with two push buttons and a PICKARRAY pick list, as shown in Figure 15-6:

Figure 15-6 SHOWDIALOG with PICKARRAY pick list



The following code creates the PICKARRAY pick list shown in Figure 15-6:

```

ARRAY RepName[7]                ; create a fixed array
RepName[1] = "Whitney"          ; assign values to array
RepName[2] = "McBride"
RepName[3] = "Thompson"
RepName[4] = "Lindsay"
RepName[5] = "Lyon"
RepName[6] = "Voss"
RepName[7] = "Ranucci"

SHOWDIALOG "Marketing Reps"     ; location and dimension of box
  @4,23 HEIGHT 13 WIDTH 30

  PICKARRAY @1,7                ; create a PICKARRAY pick list
    HEIGHT 6 WIDTH 13          ; dimensions of pick list
    RepName                    ; name of fixed array
    TAG "ArrayTag"             ; name passed to procedure as TagValue
    TO ArrayElement            ; variable assigned with tag value

  PUSHBUTTON @9,2 WIDTH 10      ; create an OK push button
    "OK"
    OK
    DEFAULT
    VALUE
    "Accept"
    TAG "AcceptTag"
    TO ButtonValue

```

```

PUSHBUTTON @9,15 WIDTH 10           ; create a CANCEL push button
"Cancel"
CANCEL
VALUE "Cancel"
TAG "CancelTag"
TO ButtonValue

```

ENDDIALOG

In the example above, notice how the PICKARRAY items are displayed in index order (not alphabetical order) in the dialog box. If the user selects the highlighted value *McBride*, the PICKARRAY variable *ArrayElement* is assigned the value 2.

---

### **PICKDYNARRAY control element**

The PICKDYNARRAY control element lets you create a pick list with the dimensions and location that you specify. The complete syntax of the PICKDYNARRAY statement follows:

```

PICKDYNARRAY
  @Row, Column
  HEIGHT Number
  WIDTH Number
  [ COLUMNS Number ]
  DynArrayName
  TAG Expression
  TO VarName

```

When you create a pick list with PICKDYNARRAY, you specify its location, height, width, and number of columns just as you do for PICKFILE. The TAG *Expression* statement is handled by PICKDYNARRAY the same way it is handled by PICKFILE.

The TO *VarName* statement is treated differently in PICKDYNARRAY than it is in PICKFILE. The values of dynamic array elements appear as items in the pick list. When a pick list item is selected, the tag of the dynamic array element is assigned to the PICKDYNARRAY variable *VarName*. For example, if *SystemArray["UIMODE"]="Standard"* is an element in a dynamic array, *Standard* appears as an item in the pick list. If this item is selected, *VarName* is assigned the value *UIMODE* when the dialog is accepted.

You can specify a pick list item as the default selection when the dialog box is opened. To specify a default, assign the PICKDYNARRAY *VarName* a value before the SHOWDIALOG statement is issued. If the value you assign is the tag of an element in *DynArrayName*, that item will appear selected when the dialog box is opened.

The values in a PICKDYNARRAY pick list are sorted and displayed alphanumerically. The following code creates a dialog box with two

push buttons and a PICKDYNARRAY pick list, as shown in Figure 15-7:

```

VIEW "C:\PDOX40\SAMPLE\CUSTOMER"           ; view the Customer table
DYNARRAY CustInfo[]                         ; create dynamic array

SCAN                                         ; scan table into dynamic array
  CustInfo[STRVAL([Cust ID])] = [Last Name] ; use Cust ID field as tag
ENDSCAN                                     ; use Last Name field as value

CLEARIMAGE                                  ; close Customer table

SHOWDIALOG "List of Customers"              ; begin dialog box definition
  @5,23 HEIGHT 14 WIDTH 31                 ; location and dimensions of box

  PICKDYNARRAY @1,3                         ; define a PICKDYNARRAY element
  HEIGHT 8 WIDTH 22                        ; dimensions of pick list box
  COLUMNS 2                                ; display pick list in two columns
  CustInfo                                  ; name of dynamic array
  TAG "DynArrayTag"                         ; name passed to procedure as TagValue
  TO CustChoice                             ; variable assigned with tag value

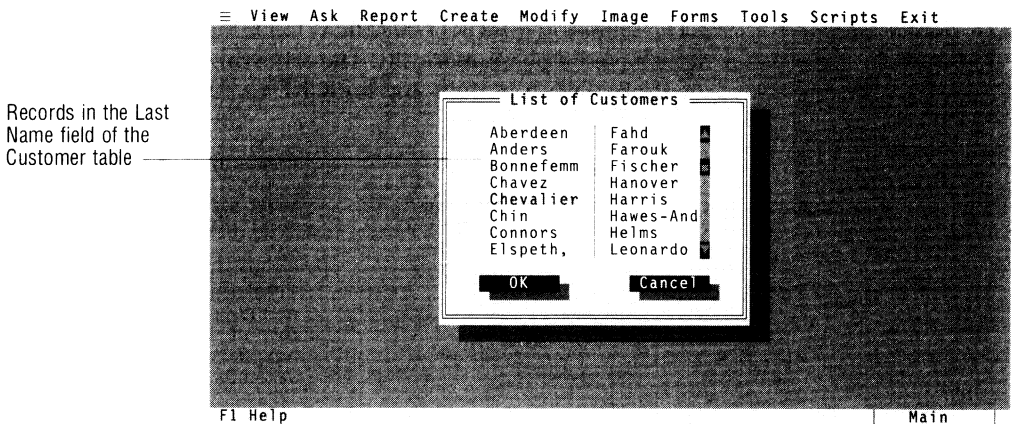
  PUSHBUTTON @10,2 WIDTH 10                ; create an OK push button
  "~O~K"
  OK
  DEFAULT
  VALUE "Accept"
  TAG "OKTag"
  TO ButtonValue

  PUSHBUTTON @10,17 WIDTH 10               ; create a Cancel push button
  "~C~ancel"
  CANCEL
  VALUE "Cancel"
  TAG "CancelTag"
  TO ButtonValue

ENDDIALOG

```

Figure 15-7 SHOWDIALOG with PICKDYNARRAY pick list



---

**PICKDYNARRAYINDEX control element**

The PICKDYNARRAYINDEX control element lets you create a pick list with the dimensions and location that you specify. The complete syntax of the PICKDYNARRAYINDEX statement follows:

```
PICKDYNARRAYINDEX  
  @Row, Column  
  HEIGHT Number  
  WIDTH Number  
  [ COLUMNS Number ]  
  DynArrayName  
  TAG Expression  
  TO VarName
```

When you create a pick list with PICKDYNARRAYINDEX, you specify its location, height, width, and number of columns just as you do for PICKDYNARRAY. The TAG Expression statement is handled by PICKDYNARRAYINDEX the same way it is handled by PICKDYNARRAY.

Unlike PICKDYNARRAY, the tags of dynamic array elements appear as items in the PICKDYNARRAYINDEX pick list. Like PICKDYNARRAY, the tag of the dynamic array element is assigned to the PICKDYNARRAYINDEX variable VarName when a pick list item is selected. For example, if SystemArray["UIMODE"]="Standard" is an element in a dynamic array, UIMODE appears as an item in the pick list. If this item is selected, VarName is also assigned the value UIMODE when the dialog is accepted.

You can specify a pick list item as the default selection when the dialog box is opened. To specify a default, assign the PICKDYNARRAYINDEX VarName a value before the SHOWDIALOG statement is issued. If the value you assign is the tag of an element in DynArrayName, that item will appear selected when the dialog box is opened.

The values in a PICKDYNARRAYINDEX pick list are sorted and displayed alphanumerically. The following code creates a dialog box with two push buttons and a PICKDYNARRAYINDEX pick list, as shown in Figure 15-8:

```
SYSINFO TO SysDynArray           ; create dynamic array with sys info  
  
SHOWDIALOG "System Info Tags"    ; begin dialog box definition  
  @5,23 HEIGHT 14 WIDTH 31      ; location and dimensions of box  
  
  PICKDYNARRAYINDEX @1,3        ; define a PICKDYNARRAYINDEX element  
  HEIGHT 8 WIDTH 22            ; dimensions of pick list box  
  COLUMNS 1                   ; display pick list in one column  
  SysDynArray                  ; name of dynamic array  
  TAG "DynArrayTag"           ; name passed to procedure as TagValue  
  TO SysChoice                 ; variable assigned with tag value
```

```

PUSHBUTTON @10,2 WIDTH 10      ; create an OK push button
  "~O~K"
  OK
  DEFAULT
  VALUE "Accept"
  TAG "OKTag"
  TO ButtonValue

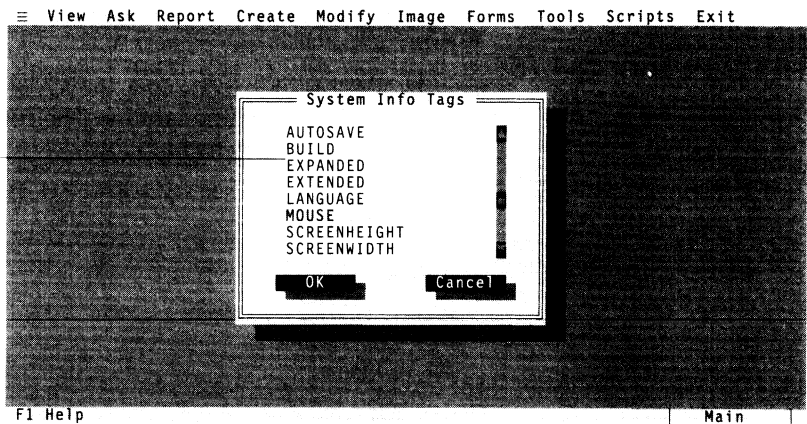
PUSHBUTTON @10,17 WIDTH 10     ; create a Cancel push button
  "~C~ancel"
  CANCEL
  VALUE "Cancel"
  TAG "CancelTag"
  TO ButtonValue

ENDDIALOG

```

Figure 15-8 SHOWDIALOG with PICKDYNARRAYINDEX pick list

Tags of elements in the dynamic array created by SYSINFO

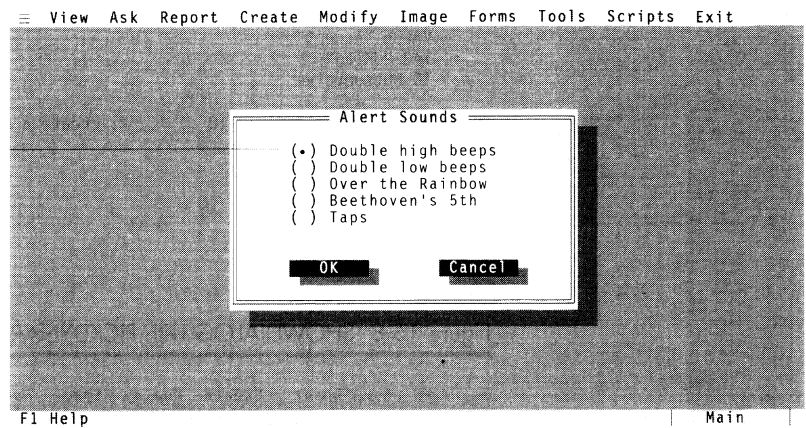


### Radio buttons

Radio buttons let the user select from *mutually exclusive* choices in a dialog box. Because no more than one radio button can be selected at any time, a bank of radio buttons is a good way to limit the user to making a single choice from a variety of available options. Compare radio buttons with check boxes, which are intended for choices that are not mutually exclusive. Figure 15-9 shows a typical SHOWDIALOG dialog box with a bank of five radio buttons.

Figure 15-9 SHOWDIALOG with radio buttons

Only a single radio button can be selected at one time



In a SHOWDIALOG dialog box, the RADIOBUTTONS statement lets you create a bank of radio buttons. The complete syntax of the RADIOBUTTONS statement follows:

#### **RADIOBUTTONS**

*@Row, Column*  
**HEIGHT** *Number*  
**WIDTH** *Number*  
*Label1, Label2, ..., LabelN*  
**TAG** *Expression*  
**TO** *VarName*

When you create a bank of radio buttons with RADIOBUTTONS, you specify the location of the upper left corner of the bank with the *@Row, Column* statement, using coordinates that are relative to the upper left corner of the dialog box. Use the HEIGHT and WIDTH keywords to specify the height and width of the area that contains the bank of radio buttons.

Each *Label* in the RADIOBUTTONS syntax is a text string that appears to the right of its radio button. The ordinal position of the radio button that is selected is assigned to the control element variable *VarName* if the dialog box is accepted. For example, if the first radio button is selected, *VarName* is assigned the value 1. The expression that follows the TAG keyword identifies the current control element for the dialog procedure.

You can specify a radio button as the default selection when the dialog box is opened. To specify a default, assign the RADIOBUTTON *VarName* a value that is the ordinal position of the desired radio button before the SHOWDIALOG statement is issued.



The following code creates the dialog box shown in Figure 15-9:

```
SHOWDIALOG "Alert Sounds"           ; define dialog box
@6,22 HEIGHT 12 WIDTH 35           ; location and dimension of box

RADIOBUTTONS @1,4 HEIGHT 5 WIDTH 25 ; define radio buttons
"Double high beeps",               ; labels next to radio buttons
"Double low beeps",
"Over the Rainbow",
"Beethoven's 5th",
"Ticks"
TAG "AlertTag"                     ; name passed to procedure as TagValue
TO AlertSelection                  ; variable assigned with tag name

PUSHBUTTON @6,4 WIDTH 10
"OK"
OK
DEFAULT
VALUE "Accept"
TAG "Yes"
TO ButtonValue

PUSHBUTTON @6,19 WIDTH 10
"Cancel"
CANCEL
VALUE "Cancel"
TAG "No"
TO ButtonValue
ENDDIALOG
```

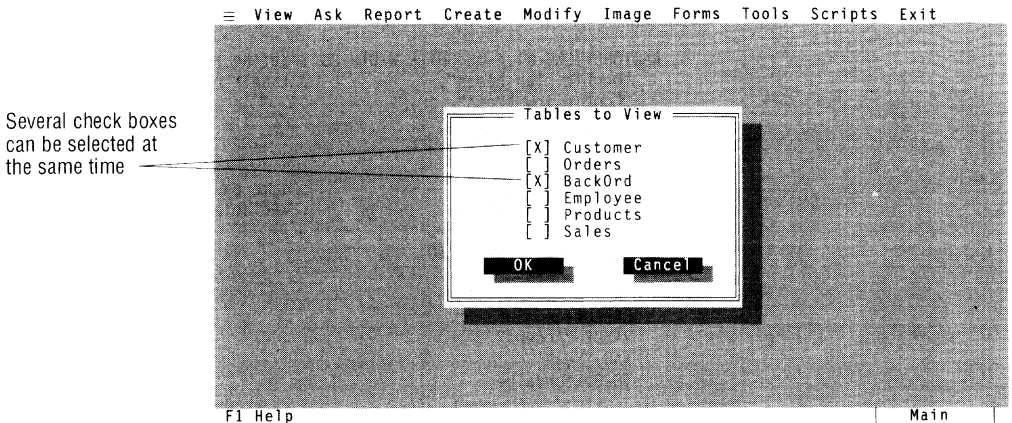
In the preceding example, notice how the entire bank of radio buttons is defined with a single `RADIOBUTTONS` statement. Each individual radio button has a separate *Label* within the statement, however. Because only a single radio button can be selected, there is only a single variable associated with the `RADIOBUTTONS` control element.

---

### **Check boxes**

Check boxes let the user select from a variety of options that are not mutually exclusive. Because more than one check box can be selected at any time, a bank of check boxes is a good way to allow the user to make several selections from a variety of related options. Compare check boxes with radio buttons, which are intended for mutually exclusive choices. Figure 15-10 shows a typical `SHOWDIALOG` dialog box with a bank of six check boxes.

Figure 15-10 SHOWDIALOG with check boxes



In a SHOWDIALOG dialog box, the CHECKBOXES statement lets you create a bank of check boxes. The complete syntax of the CHECKBOXES statement follows:

#### CHECKBOXES

*@Row, Column*  
**HEIGHT** *Number*  
**WIDTH** *Number*  
**TAG** *Expression*  
*Label1 TO VarName1,*  
*Label2 TO VarName2,*  
*...*  
*LabelN TO VarNameN*

Specify the location and dimensions of the bank of check boxes with the *@ Row, Column*, **HEIGHT**, and **WIDTH** statements just as you do with **RADIOBUTTONS**. Like **RADIOBUTTONS**, each *Label* is a text string that appears to the right of its check box. Unlike **RADIOBUTTONS**, each *Label* in **CHECKBOXES** is associated with a variable. If the check box is selected when the dialog box is accepted, its associated variable is assigned the value True. For example, if *Label1* and *Label2* are both selected, *VarName1* and *VarName2* are both assigned the value True. The expression that follows the **TAG** keyword identifies the current control element for the dialog procedure.

You can specify check boxes to be selected or deselected by default when the dialog box is opened. To select a check box by default, assign the **CHECKBOX** *VarName* the value True before the **SHOWDIALOG** statement is issued.

The following code creates the dialog box shown in Figure 15-10:

```

TableDir = "C:\\PDOX40\\SAMPLE\\" ; directory where tables reside

ARRAY TabNames[6] ; create fixed array TabNames
TabNames[1] = "Customer" ; load array with table names
TabNames[2] = "Orders"
TabNames[3] = "BackOrd"
TabNames[4] = "Employee"
TabNames[5] = "Products"
TabNames[6] = "Sales"

ARRAY CheckBox[ARRAYSIZE(TabNames)]

CheckBox[1] = True ; 1st check box is selected by default

SHOWDIALOG "Tables to View" ; location and dimensions of dialog box
@6,23 HEIGHT 12 WIDTH 30

CHECKBOXES @1,6 HEIGHT 6 WIDTH 16
TAG "Check" ; name passed to procedure as TagValue
TabNames[1] TO CheckBox[1], ; each check box is assigned the
TabNames[2] TO CheckBox[2], ; value True if it is selected
TabNames[3] TO CheckBox[3],
TabNames[4] TO CheckBox[4],
TabNames[5] TO CheckBox[5],
TabNames[6] TO CheckBox[6]

PUSHBUTTON @6,2 WIDTH 10 ; create an OK push button
"~O~k"
OK
DEFAULT
VALUE "Accept"
TAG "Yes"
TO ButtonValue

PUSHBUTTON @6,16 WIDTH 10 ; create a Cancel push button
"~C~ancel"
CANCEL
VALUE "Cancel"
TAG "No"
TO ButtonValue

ENDDIALOG

IF Retval ; if dialog box was accepted
THEN FOR i FROM 1 TO 6 ; view each table that
IF CheckBox[i] = True ; was checked
THEN VIEW TableDir + TabNames[i]
ENDIF
ENDFOR
ENDIF

```

In the preceding example, notice how the entire bank of check boxes is defined with a single CHECKBOXES statement. Each individual check box has a separate *Label* within the statement, however. Because more than one check box can be selected, there is a variable associated with each *Label* in the CHECKBOXES control element.

When a dialog box is first opened, each check box is initially selected or deselected based on the value of its variable. If the value of a check box variable is True, the check box appears selected when the dialog box is opened. In the example above, the Customer check box is initially selected because the value of CheckBox[1] is initially True.

---

## Type-in boxes

Type-in boxes are used to accept keyboard input from the user. You can create a type-in box within a SHOWDIALOG by using the ACCEPT statement. Figure 15-11 shows a typical SHOWDIALOG dialog box with a type-in box. The complete syntax of the ACCEPT statement follows:

### ACCEPT

```
@Row, Column
WIDTH Number
DataType
[ PICTURE Picture ][ MIN Value ][ MAX Value ]
[ LOOKUP TableName ][ REQUIRED ][ HIDDEN ]
TAG Expression
TO VarName
```

Specify the location of the left side of the type-in box with the @ *Row*, *Column* statement, using coordinates that are relative to the upper left corner of the dialog box. Use the WIDTH keyword to specify the number of characters the type-in box will accept before scrolling. The total number of characters the type-in box will accept is determined by the specified *DataType*. The optional HIDDEN keyword lets you hide the text that is typed in, such as passwords. The expression that follows the TAG keyword identifies the current control element for the dialog procedure.

Unlike the ACCEPT command, the ACCEPT control element does not use a DEFAULT keyword; the initial value (if any) of the *VarName* variable provides the default. The rest of the syntax for ACCEPT is the same as the syntax for the ACCEPT command that is used outside a dialog box. See the ACCEPT command description in the *PAL Reference* for complete information.

The following code creates the simple dialog box shown earlier in this chapter in Figure 15-3:

```
TheName = "Type here"           ; initialize the ACCEPT variable

SHOWDIALOG "Name" @ 6, 15 HEIGHT 10 WIDTH 50
@ 2, 8 ?? "Enter your name:"   ; display a prompt on the dialog box

ACCEPT @ 2, 25                 ; specify location of type-in box
WIDTH 15 "A10"                 ; specify width of type-in box
TAG "UserName"                 ; tag passed to dialog proc
TO TheName                     ; ACCEPT variable initialized above

PUSHBUTTON @5,9 WIDTH 10       ; create an OK push button
"OK"
OK
DEFAULT
VALUE "Say Yes"
TAG "Yes"
TO ButtonValue

PUSHBUTTON @5,29 WIDTH 10      ; create a Cancel push button
"Cancel"
```

```

CANCEL
VALUE "Say No"
TAG "No"
TO ButtonValue

```

```

ENDDIALOG

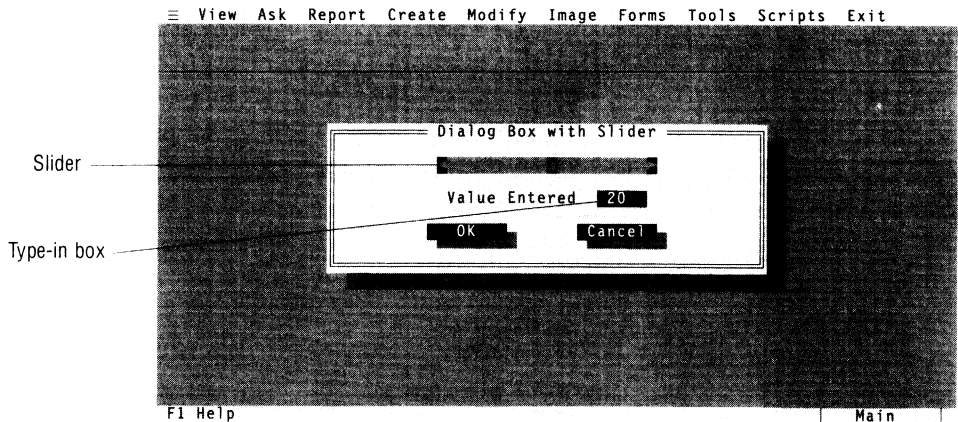
```

In the preceding example, notice how the ACCEPT control variable *TheName* is initialized to the value of *Type here* at the beginning of the script. When the dialog box is initially displayed, the type-in box displays the initialized value *Type here* and selects it. Because this ACCEPT statement specifies a 10-character data element and a 15-character width, the type-in box will not scroll.

## Sliders

The SLIDER command is used to display a slider control within a dialog box. The slider control has arrows on either end, a scroll bar, and a scroll box, but it is *not* the same as a scroll bar that appears on a window or a pick list. Paradox creates scroll bars on windows and pick lists automatically when they are needed; a slider is a separate element that you can create within a dialog box by using PAL. Figure 15-11 shows a typical SHOWDIALOG dialog box with a horizontal slider and a type-in box.

Figure 15-11 SHOWDIALOG with slider and type-in box



The complete syntax of the SLIDER statement follows:

```

SLIDER
  @Row, Column
  { VERTICAL | HORIZONTAL }
  LENGTH Number
  MIN Value
  MAX Value
  ARROWSTEP Value

```

**PAGESTEP** *Value*  
**TAG** *Expression*  
**TO** *VarName*

Specify the location of the upper left corner of the slider with the @ *Row, Column* statement, using coordinates that are relative to the upper left corner of the dialog box. Use the VERTICAL or HORIZONTAL keywords to specify an orientation for the slider, and the LENGTH keyword to specify the length of the slider in characters.

The MIN and MAX keywords establish a minimum and maximum value for the slider, respectively. The ARROWSTEP and PAGESTEP keywords specify the amount the slider value changes when the scroll arrow or scroll bar is clicked. *VarName* is assigned the value of the slider when the dialog is accepted. The expression that follows the TAG keyword identifies the current control element for the dialog procedure.

You can specify the default position of the slider when the dialog box is opened. To specify the slider position, assign the SLIDER *VarName* a value before the SHOWDIALOG statement is issued.

The following code creates the slider element for the dialog box shown in Figure 15-11:

```
ControlVar = 20 ; optional initial value for
                  ; SLIDER and ACCEPT variable
SHOWDIALOG "Dialog Box with Slider"
  @7,17 HEIGHT 9 WIDTH 44 ; location and dimensions of dialog box

  SLIDER @1,10 HORIZONTAL LENGTH 22 ; create a slider
    MIN 0 ; restrict input between 0 and 40
    MAX 40
    ARROWSTEP 1 ; slider changes 1 for each arrow click
    PAGESTEP 5 ; slider changes 5 for each page click
    TAG "SliderTag" ; name passed to procedure as TagValue
    TO ControlVar ; variable assigned with tag value

  ACCEPT @3,26 WIDTH 5 ; create an ACCEPT field
    "N"
    MIN 0
    MAX 40
    TAG "AcceptTag"
    TO ControlVar ; same variable used for SLIDER and ACCEPT

  PUSHBUTTON @5,8 WIDTH 10 ; create an OK push button
    "~O~K"
    OK
    DEFAULT
    VALUE "Accept"
    TAG "Yes"
    TO ButtonValue

  PUSHBUTTON @5,23 WIDTH 10 ; create a Cancel push button
    "~C~ancel"
    CANCEL
    VALUE "Cancel"
    TAG "No"
    TO ButtonValue
ENDDIALOG
```

In the preceding example, notice how the variable *SliderValue* is initialized at the beginning of the script. When the dialog box is initially displayed, the value of the slider is 20. Because *ARROWSTEP* is set to 1, the value of the slider changes by one if the scroll arrow or keyboard arrow is used. Because *PAGESTEP* is set to 5, the value of the slider changes by five if the scroll bar (or the keyboard arrow with *Ctrl*) is used.

As Figure 15-11 shows, a slider control is frequently used with a type-in box. See “Using the *RESYNCCONTROL* command” later in this chapter for information about making the slider control and the type-in box work together.

---

## Labels

A label is not a separate dialog box control element; rather, it is a hot key that can be attached to any existing dialog box control element. When the label is clicked or the hot key is pressed, the element linked to the label becomes the active control element in the dialog box. In Figure 15-11, the text string *Value Entered* is a label for the *ACCEPT* control; the letter *V* is the hot key for the label.

In a *SHOWDIALOG* dialog box, the *LABEL* statement lets you create a label. The complete syntax of the *LABEL* statement follows:

### LABEL

```
@Row,Column  
LabelName  
FOR TagExpression
```

When you create a label, you specify its location with the *@ Row, Column* statement in coordinates that are relative to the upper left corner of the dialog box. Specify a name for the label with *LabelName*, and create a hot key in the label by surrounding the hot key with tilde (~) characters. The *TagExpression* that you specify must be the same *Expression* used as a *TAG* in the control element that the label is linked to.

For example, the following code creates a label for the *ACCEPT* control in the dialog box shown in Figure 15-11:

```
ControlVar = 20 ; optional initial value for  
; SLIDER and ACCEPT variable  
SHOWDIALOG "Dialog Box with Slider"  
@7,17 HEIGHT 9 WIDTH 44 ; location and dimensions of dialog box  
  
SLIDER @1,10 HORIZONTAL LENGTH 22 ; create a slider  
MIN 0  
MAX 40  
ARROWSTEP 1  
PAGESTEP 5  
TAG "SliderTag"  
TO ControlVar  
  
LABEL @3,10 ; create a label called  
"~V-alue Entered" ; Value Entered with V as a hot key
```

```

FOR "AcceptTag"                ; TAG of control element that label
                                ; is associated with

ACCEPT @3,26 WIDTH 5           ; create an ACCEPT field
  "N"
  MIN 0
  MAX 40
  TAG "AcceptTag"              ; same TAG is used in LABEL clause
  TO ControlVar

PUSHBUTTON @5,8 WIDTH 10       ; create an OK push button
  "~O~K"
  OK
  DEFAULT
  VALUE "Accept"
  TAG "Yes"
  TO ButtonValue

PUSHBUTTON @5,23 WIDTH 10      ; create a Cancel push button
  "~C~ancel"
  CANCEL
  VALUE "Cancel"
  TAG "No"
  TO ButtonValue

ENDDIALOG

```

---

## Using canvas elements to control dialog box background

Canvas elements let you control the appearance of a dialog box; you do not need to specify any canvas elements to create a valid dialog box. The SHOWDIALOG dialog box lets you control the following canvas elements:

- one or more string expressions for background text
- location and style for background text
- single or double frames in specific locations
- colors for any area of the background

Following is the complete syntax for SHOWDIALOG *CanvasElements*:

```

[ STYLE [ MonoOptionList ] | STYLE ATTRIBUTE Number ]
[ @Row, Column ]
[ ? ExpressionList ]
[ ?? ExpressionList ]
[ FRAME [ SINGLE | DOUBLE ] FROM Row1, Col1 TO Row2, Col2 ]
[ CLEAR [ EOL | EOS ] ]
[ PAINTCANVAS [ BORDER ] [ FILL String ]
  { MonoOptionList | ATTRIBUTE Number | BACKGROUND }
  { Row1, Column1, Row2, Column2 | ALL } ]

```

The canvas elements that are available within a dialog box are the same as the PAL commands that are available to control writing to a canvas. For example, the PAINTCANVAS clause within a SHOWDIALOG statement has the same keywords and parameters as the general PAINTCANVAS command. The coordinates that you



specify for all canvas elements are relative to the dialog box, like the coordinates for control elements.

The following code uses a variety of canvas-writing commands to create the dialog box shown in Figure 15-12:

```

SHOWDIALOG "Canvas Elements"      ; show dialog box with title
@ 2,6 HEIGHT 13 WIDTH 65         ; location and dimension of dialog box

FRAME SINGLE FROM 1, 1 TO 9, 61 ; place a frame in the dialog box

PAINTCANVAS BORDER ATTRIBUTE (15 + 16) 1, 1, 9, 61
                                ; BORDER affects a 1-column rectangle
                                ; attribute 15 + 16 is white on blue
                                ; location is on top of the frame so
                                ; the frame colors are white on blue

PAINTCANVAS BORDER FILL CHR(177) 2, 2, 8, 60
                                ; BORDER affects a 1-column rectangle
                                ; fill specifies a character fill
                                ; location is within the frame

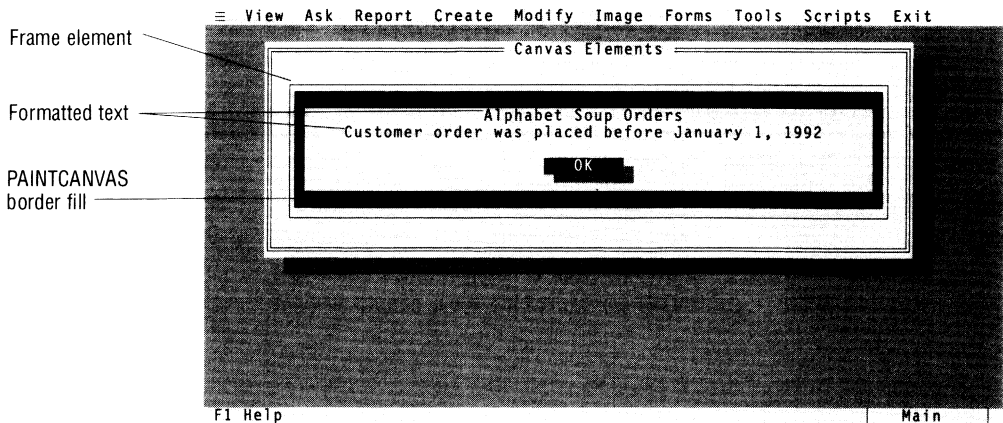
STYLE ATTRIBUTE 15 + 16          ; text colors will be white on blue
@ 3,3 ?? FORMAT("W57,AC", "Alphabet Soup Orders")
@ 4,3 ?? FORMAT("W57,AC", "Customer order was placed before January 1, 1992")

PUSHBUTTON @6,26 WIDTH 10       ; create a push button
"OK"
OK
DEFAULT
VALUE "Yes"
TAG "ACCEPT"
TO ButtonValue

ENDDIALOG

```

Figure 15-12 SHOWDIALOG canvas elements



## Using the dialog procedure

If you use a dialog procedure with the SHOWDIALOG command, a user can interact with a dialog until one of the events specified in the *EventList* occurs. When the specified event occurs, the *DialogProc*

procedure is called. The *DialogProc* is typically used to change the dialog box in response to a user interaction, although it is not limited to this use.

For example, Figure 15-11 shows a dialog box with a slider and a type-in box. If you *synchronize* these two controls, changes to the slider position will be reflected in the type-in box; new values entered in the type-in box will update the position of the slider. The SHOWDIALOG command can trap for events that are issued when one of these controls is updated, and then use the dialog procedure to update the other control.

---

**Structure of the dialog procedure**

The dialog procedure can be called for any mouse, key, message, or idle event, or any of the triggers listed in Table 13-7. The following discussion explains how to take advantage of event-driven programming within a SHOWDIALOG dialog box. If you are uncertain about how to use events or triggers, you should review Chapter 13, "Handling events."

The dialog procedure *DialogProc* takes four arguments to help you determine the exact situation in which it was called. You can name these arguments differently when you use them; for the purposes of this manual, we have named them as follows:

*DialogProc* (*TriggerType*, *TagValue*, *EventValue*, *ElementValue*)

The values that are assigned to the *DialogProc* arguments are determined by a combination of the calling event or trigger and the control element that is active when the *DialogProc* procedure is called.

If the *DialogProc* procedure is called for a mouse, key, message, or idle event, the *DialogProc* arguments are assigned values as described in Table 15-1:

Table 15-1 Assignments to dialog procedure arguments made by events

Event	TriggerType	TagValue	EventValue	ElementValue
Mouse	EVENT	TAG <i>Expression</i> of the active control element	Dynamic array	Null string
Key	EVENT	TAG <i>Expression</i> of the active control element	Dynamic array	Null string
Message	EVENT	TAG <i>Expression</i> of the active control element	Dynamic array	Null string
Idle	EVENT	TAG <i>Expression</i> of the active control element	Dynamic array	Null string

The dynamic array created for the *EventValue* argument when the *DialogProc* is called for an event is the same as the dynamic array that GETEVENT would create for that event.

If the *DialogProc* procedure is called for a trigger, the values that are passed to the *DialogProc* arguments depend on the type of trigger and the control element that was active when the *DialogProc* was called. Table 15-2 summarizes the assignments of these arguments.

Table 15-2 Assignments to dialog procedure arguments made by triggers

Trigger	TriggerType	TagValue	EventValue	ElementValue
UPDATE	UPDATE	Value of the TAG <i>Expression</i> of the active control element	Current value of the active control element variable <i>VarName</i>	Label string for a CHECKBOX control element; otherwise null string
DEPART	DEPART	Value of the TAG <i>Expression</i> of the active control element	TAG <i>Expression</i> of the control element you will navigate to	Null string
SELECT	SELECT	Value of the TAG <i>Expression</i> of the active control element	Null string	Null string
ACCEPT	ACCEPT	Value of the TAG <i>Expression</i> of the active control element	Null string	Null string
CANCEL	CANCEL	Value of the TAG <i>Expression</i> of the active control element	Null string	Null string
ARRIVE	ARRIVE	Value of the TAG <i>Expression</i> of the active control element	Null string	Null string
OPEN	OPEN	Value of the TAG <i>Expression</i> of the active control element	Null string	Null string
CLOSE	CLOSE	Value of the TAG <i>Expression</i> of the active control element	Null string	Null string

The event or trigger that the dialog procedure is called for is still *pending* when the dialog procedure terminates. If you return the value True from the dialog procedure, the pending event is passed on to Paradox and processed normally. If you return the value False from the dialog procedure, the pending event is *denied*; that is, it is not passed on to Paradox and processed. If you do not return a value, the pending event is processed normally.

The best way to learn about event processing within a dialog box is to trap events or triggers and use the *DialogProc* procedure to display the resulting variable assignments onscreen. Example 15-1 shows how you can use the *DialogProc* procedure to display this information about a typical dialog box. Figure 15-13 shows the value of the *DialogProc* procedure arguments for a typical SHOWDIALOG dialog box.

### Example 15-1 Dialog procedure arguments and trigger sequence

In this SHOWDIALOG example, the *DialogProc* procedure is called for all triggers. The values of the four *DialogProc* arguments are displayed on a canvas; as you scroll through the pick list, the new values of the arguments are displayed on the canvas.

```
ECHO NORMAL ; show user the window
WINDOW CREATE @2,3 WIDTH 40 HEIGHT 8 TO DisplayWindow ; create a window
DYNARRAY DWinAttrib[] ; create dynamic array
DWinAttrib["TITLE"] = "Variables and their values" ; specify title element
WINDOW SETATTRIBUTES DisplayWindow FROM DWinAttrib ; change attributes

SETMARGIN 2 ; set left margin offset
? "TriggerType" ; write variable names in new window
? "TagValue"
? "EventValue"
? "ElementValue"
SETMARGIN 17 ; subsequent text will not overwrite
; variable names

; define the dialog procedure
PROC ShowVars(TriggerType, TagValue, EventValue, ElementValue)
@0,17 CLEAR EOS ; clear any residual values
? "= ", TriggerType ; write variable values
? "= ", TagValue
? "= ", EventValue
? "= ", ElementValue
SLEEP 1000 ; pause to view the canvas window
RETURN True ; process the event
ENDPROC

SHOWDIALOG "List of Tables" ; begin definition
PROC "ShowVars" ; procedure to execute
TRIGGER "ALL" ; trap for all triggers
@7,46 HEIGHT 14 WIDTH 31 ; location and dimension of dialog

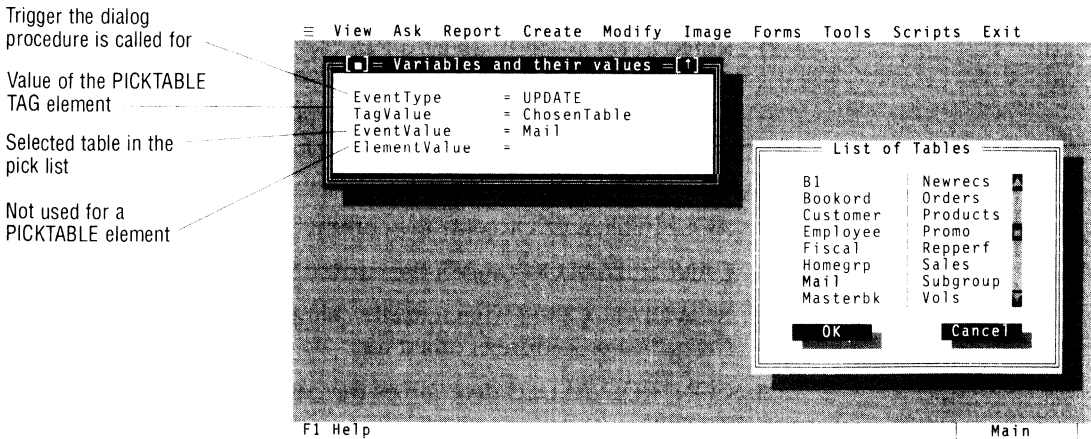
PICKTABLE @1,3 HEIGHT 8 WIDTH 22 ; define a pick list
COLUMNS 2
"C:\PDOX40\SAMPLE\"
TAG "ChosenTable"
TO TableChoice

PUSHBUTTON @10,2 WIDTH 10 ; define a push button
"-O~K"
OK
DEFAULT
VALUE "Accept"
TAG "Yes"
TO ButtonValue

PUSHBUTTON @10,17 WIDTH 10 ; define a push button
"-C~ancel"
CANCEL
VALUE "Cancel"
TAG "No"
TO ButtonValue

ENDDIALOG
```

Figure 15-13 Typical values of dialog procedure arguments



In the above example, the *DialogProc* procedure is called for every trigger that occurs. When you first run this script, an OPEN trigger indicates that a SHOWDIALOG command has been issued. If you select a pick list item with the mouse or keyboard, the canvas window display changes to indicate that an UPDATE trigger has called the *DialogProc* procedure. If you double-click a pick list item or press the *Space* key, the result is a SELECT trigger. Clicking the OK button results in a series of triggers issued in this order: DEPART, ARRIVE, UPDATE, ACCEPT, and CLOSE.

In this example, the *DialogProc* procedure simply causes the canvas window to display the values of the four *DialogProc* arguments. An actual application could use the *DialogProc* procedure to take an action such as viewing or deleting the selected table.

## Using commands and functions to control the dialog box

PAL provides a special set of commands and functions to help you control dialog box interactions. The following commands are all available for use in the *DialogProc* procedure or in a procedure called by the *DialogProc* procedure:

- REPAINTDIALOG executes each canvas painting command again, recalculating dynamic expressions and redrawing the current SHOWDIALOG dialog box.
- RESYNCCONTROL examines the value of a control element variable and displays the new value for that control element onscreen.
- RESYNCDIALOG performs a RESYNCCONTROL for each control element within the current SHOWDIALOG dialog box and issues a REPAINTDIALOG.

- ❑ SELECTCONTROL navigates within the current SHOWDIALOG dialog box.
- ❑ REFRESHCONTROL issues a RESYNCCONTROL and also updates the appearance and function of the control.
- ❑ REFRESHDIALOG performs a REFRESHCONTROL for each control element within the current SHOWDIALOG dialog box and issues a REPAINTDIALOG.
- ❑ ACCEPTDIALOG accepts the current SHOWDIALOG dialog box.
- ❑ CANCELDIALOG cancels the current SHOWDIALOG dialog box.
- ❑ NEWDIALOGSPEC dynamically changes the event list that the *DialogProc* procedure is called for.
- ❑ CONTROLVALUE() determines the current value of any control element.

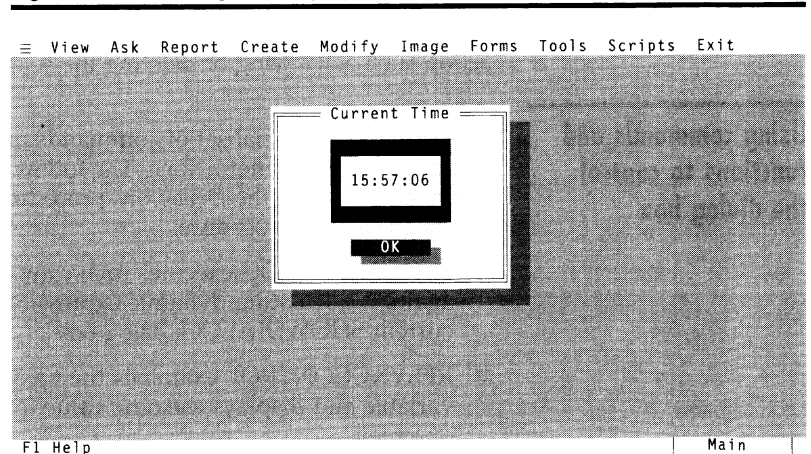
The remaining sections of this chapter discuss these commands and the dialog procedure in detail.

---

**Using the REPAINTDIALOG command**

The REPAINTDIALOG command executes each canvas painting command again, recalculating all dynamic expressions within a dialog box and repainting the screen display. REPAINTDIALOG does not affect any control elements. The REPAINTDIALOG command is useful for repainting the screen when the value of a dialog expression changes.

Figure 15-14 Dialog box displaying current system time



For example, the dialog box shown in Figure 15-14 uses the TIME() function to display the current system time. Because the current system time is constantly changing, the dialog box display must be

repainted constantly. The following example shows how REPAINTDIALOG could be used in this situation:

```
PROC RepaintProc(TriggerType, TagValue, EventValue, ElementValue)
  REPAINTDIALOG ; repaint the dialog canvas
  RETURN True ; process the event
ENDPROC

SHOWDIALOG "Current Time" ; show dialog box with title
PROC "RepaintProc" IDLE ; execute proc on IDLE
@ 4,26 HEIGHT 11 WIDTH 24 ; location and dimension of box

@ 3,6 ?? FORMAT("W10,AC",TIME()) ; display the current time
PAINTCANVAS BORDER FILL CHR(219) 1,5,5,16 ; place a border around time
PAINTCANVAS ATTRIBUTE 15 + 16 2,6,4,15 ; box interior white on blue

PUSHBUTTON @7,6 WIDTH 10 ; place an OK button
  "~0~k"
  OK
  DEFAULT
  VALUE "Yes"
  TAG "OKTag"
  TO ButtonValue

ENDDIALOG
```

In the above example, the dialog procedure is called for every idle event, which means the the dialog procedure is called at frequent regular intervals. The dialog procedure in this example is used only to repaint the dialog, so the time display is always accurate.

---

### **Using the RESYNCCONTROL command**

The RESYNCCONTROL command resynchronizes a specified control element with the current value of its *VarName* variable. For example, when you want to change the value of a control element after a dialog box has been opened, you could change the value of the *VarName* variable used by that control element. You then could issue a RESYNCCONTROL command to make the control element that the user sees reflect the new value of its variable.

As mentioned previously, you often want a slider and a type-in box such as the ones shown in Figure 15-11 to reflect the same value. To synchronize these control elements, you could call the dialog procedure for the UPDATE triggers that occur when a control element is changed. Use the RESYNCCONTROL command within the dialog procedure to update the slider if the type-in box changes, and to update the type-in box if the slider changes.

For example, adding a dialog procedure to the code shown in the "Labels" section could synchronize the type-in box and the slider shown in Figure 15-11. You would, of course, also have to change the SHOWDIALOG command to call the dialog procedure:

```
PROC SliderProc(TriggerType, TagValue, EventValue, ElementValue)
  IF TriggerType = "UPDATE" ; make sure trigger type is UPDATE
  THEN
    IF TagValue = "SliderTag" OR TagValue = "AcceptTag"
      ; if SLIDER or ACCEPT has focus
```

```

        THEN ControlVar = EventValue ; then assign ACCEPT and SLIDER vars
          RESYNCCONTROL "SliderTag" ; to value of new selection and
          RESYNCCONTROL "AcceptTag" ; update SLIDER and ACCEPT controls
          RETURN True ; process the UPDATE trigger
      ENDIF
    ENDIF
    RETURN True ; process the update event
  ENDPROC

ControlVar = 20

SHOWDIALOG "Dialog Box with Slider"
PROC "SliderProc" TRIGGER "UPDATE" ; trap for UPDATE triggers
@7,17 HEIGHT 9 WIDTH 44

SLIDER @1,10 HORIZONTAL LENGTH 22 ; create a slider
MIN 0
MAX 40
ARROWSTEP 1
PAGESTEP 5
TAG "SliderTag"
TO ControlVar

LABEL @3,10 ; create a label
"--Value Entered"
FOR "AcceptTag"

ACCEPT @3,26 WIDTH 5 ; create an ACCEPT type-in box
"N"
MIN 0
MAX 40
TAG "AcceptTag"
TO ControlVar

PUSHBUTTON @5,8 WIDTH 10 ; create an OK push button
"--O-K"
OK
DEFAULT
VALUE "Accept"
TAG "Yes"
TO ButtonValue

PUSHBUTTON @5,23 WIDTH 10 ; create a Cancel push button
"--C-ancel"
CANCEL
VALUE "Cancel"
TAG "No"
TO ButtonValue

ENDDIALOG

```

The dialog procedure in the above example is called for an UPDATE trigger. When the user interacts with either the SLIDER or ACCEPT control, the IF control structure executes. The IF control structure sets the value of the *ControlVar* variable equal to the *EventValue* argument, which contains the current value of the active control. Because *ControlVar* is used for both the SLIDER and the ACCEPT, both control elements now have the same value. The RESYNCCONTROL commands then cause the SLIDER and the ACCEPT control elements to reflect this value; the control elements are linked together. The control structure then returns the value True so Paradox processes the trigger.



The RESYNCCONTROL command has a different effect for each control element. Do not confuse the RESYNCCONTROL command with the REFRESHCONTROL command; REFRESHCONTROL issues an implicit RESYNCCONTROL. The effects of both commands are compared and summarized in Table 15-3.

Table 15-3 RESYNCCONTROL actions vs. REFRESHCONTROL actions

Control element	RESYNCCONTROL action	REFRESHCONTROL action
ACCEPT	Uses the value of the TO variable to update the string that appears in the type-in box	Updates the DataType element and the values assigned in the PICTURE; uses the value of the TO variable to update the string that appears in the type-in box
PUSHBUTTON	No effect	No effect
RADIOBUTTON	Highlights the radio button element specified by the value of the TO variable	Highlights the radio button element specified by the value of the TO variable
CHECKBOX	Highlights the check box elements specified by the values of the TO variables	Highlights the check box elements specified by the values of the TO variables
LABEL	No effect	No effect
PICKFILE	Highlights the pick list element that most closely matches the value of the TO variable	Generates a new pick list by reevaluating the file specification in the Path statement; highlights the pick list element that most closely matches the value of the TO variable
PICKARRAY	Highlights the pick list element that most closely matches the value of the TO variable	Generates a new pick list by reevaluating the contents of the fixed array; highlights the pick list element that most closely matches the value of the TO variable
PICKDYNARRAY	Highlights the pick list element that most closely matches the value of the TO variable	Generates a new pick list by reevaluating the contents of the dynamic array; highlights the pick list element that most closely matches the value of the TO variable

Control element	RESYNCCONTROL action	REFRESHCONTROL action
PICKDYNARRAYINDEX	Highlights the pick list element that most closely matches the value of the TO variable	Generates a new pick list by reevaluating the contents of the dynamic array; highlights the pick list element that most closely matches the value of the TO variable
PICKTABLE	Highlights the pick list element that most closely matches the value of the TO variable	Generates a new pick list by reevaluating the file specification in the Path statement; highlights the pick list element that most closely matches the value of the TO variable
SLIDER	Moves the scroll box to the location specified by the TO variable	Updates the values assigned in the MIN, MAX, ARROWSTEP, and PAGESTEP statements; moves the scroll box to the location specified by the TO variable

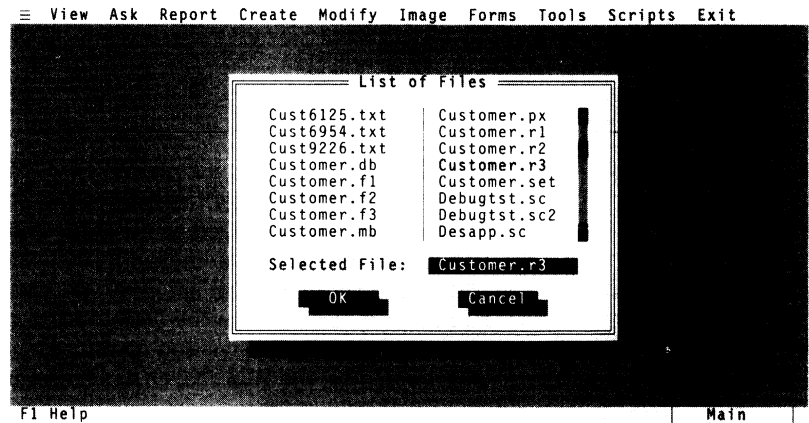
### ***Using the RESYNCDIALOG command***

When you want to resynchronize all the control elements within a dialog box, you can use the RESYNCDIALOG command. The RESYNCDIALOG command resynchronizes every control element in the active dialog box with the current value of its *VarName* variable and implicitly issues a REPAINTDIALOG. Because RESYNCCONTROL does not result in a REPAINTDIALOG, the RESYNCDIALOG command is slightly different than issuing a RESYNCCONTROL for every control element.

When you change the value of more than one control element variable, you could use RESYNCDIALOG to make each control element that the user sees reflect the new value of its variable, rather than issuing a RESYNCCONTROL for every control element. However, you could gain a performance advantage by simply issuing multiple RESYNCCONTROL commands and avoiding the REPAINTDIALOG that RESYNCDIALOG implies.

For example, you could use RESYNCDIALOG to make sure that the values in a pick list and a type-in box, such as the ones shown in the dialog box in Figure 15-15, are always the same.

Figure 15-15 Pick list and type-in box values are the same



The RESYNCCONTROL example created a similar effect by using a separate RESYNCCONTROL command for each control element. You could also use a single RESYNCDIALOG as follows:

```
PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
    ; proc is called only on update trigger

    IF TagValue = "PickTag" OR
       TagValue = "AcceptTag"
       THEN FileName = EventValue
           RESYNCDIALOG
           ; if pickfile list is updated or
           ; if accept field is updated
           ; then assign ACCEPT and PICKFILE vars
           ; to value of new selection and
           ; update PICKFILE and ACCEPT controls
           ; since their variables have changed
    ENDIF
    RETURN True
ENDPROC
; process the update event

SHOWDIALOG "List of Files"
PROC "DialogProc"
TRIGGER "UPDATE"
@4,22 HEIGHT 16 WIDTH 39

@ 10,3 ?? "Selected File:"

PICKFILE @1,2 HEIGHT 8 WIDTH 32
COLUMNS 2
DIRECTORY() + "*,*"
TAG "PickTag"
TO FileName

ACCEPT @10,19 WIDTH 15
"A12"
TAG "AcceptTag"
TO FileName

PUSHBUTTON @12,5 WIDTH 10
"~O~K"
OK
DEFAULT
VALUE "Accept"
```

```

TAG "OKTag"
TO ButtonValue

PUSHBUTTON @12,21 WIDTH 10
"~C~ancel"
CANCEL
VALUE "Cancel"
TAG "CancelTag"
TO ButtonValue

ENDDIALOG

```

In the RESYNCCONTROL example, the SLIDER and the ACCEPT control were synchronized to the same value by issuing two RESYNCCONTROL statements. In the example above, the PICKFILE and the ACCEPT control are given the same value by the RESYNCDIALOG when the *DialogProc* is called. If this dialog box contained any canvas-writing commands, however, they would also be executed by the RESYNCDIALOG command.

---

### **Using the SELECTCONTROL command**

The SELECTCONTROL command is used to control the location of the cursor within the dialog box. You can navigate with SELECTCONTROL by specifying the TAG name of the control element you want the cursor positioned on. SELECTCONTROL does not generate DEPART, UPDATE, or ARRIVE triggers and does not cause the *DialogProc* procedure to be called. The SELECTCONTROL command positions the cursor on the specified control and stops further navigation.

For example, you could use SELECTCONTROL to move the cursor to a specific control element in the dialog box, forcing the user to interact with that control in certain situations. The following code uses SELECTCONTROL to move the cursor to an ACCEPT control element if the user tries to accept a dialog box without first entering a value in the type-in box:

```

PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
; proc is called only when
; dialog is accepted
IF ISBLANK(IDNumber) ; if user left type-in box empty
THEN MESSAGE "Enter credit card number"
SLEEP 1500
MESSAGE ""
SELECTCONTROL "AcceptTag" ; place cursor on type-in box
RETURN False ; deny the accept trigger
ELSE RETURN True ; if type-in box isn't empty
; process the accept trigger
ENDIF
ENDPROC

PayMethod = 1 ; select first radio button as default

SHOWDIALOG "Method of payment"
PROC "DialogProc" TRIGGER "ACCEPT" ; trap for accept trigger
@6,22 HEIGHT 14 WIDTH 35

RADIOBUTTONS @1,4 HEIGHT 4 WIDTH 25 ; create a bank of radio buttons

```

```

"Visa",
"MasterCard",
"American Express",
"Diners Club"
TAG "PayType"
TO PayMethod

LABEL @6,6 ; create a label
"-E~nter credit card#/"
FOR "AcceptTag"

ACCEPT @7,6 WIDTH 21 ; create an ACCEPT type-in box
"A10"
TAG "AcceptTag"
TO IDNumber

PUSHBUTTON @10,4 WIDTH 10 ; create an ACCEPT type-in box
"OK"
OK
DEFAULT
VALUE "Accept"
TAG "OKTag"
TO ButtonValue

PUSHBUTTON @10,19 WIDTH 10 ; create an ACCEPT type-in box
"Cancel"
CANCEL
VALUE "Cancel"
TAG "CancelTag"
TO ButtonValue

ENDDIALOG

```

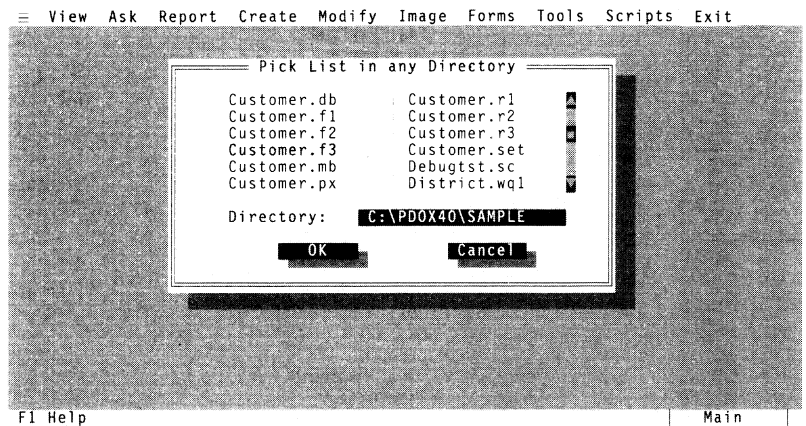
In the above example, the dialog procedure is called for the ACCEPT trigger that occurs when a user tries to accept a dialog box. The IF control structure in the dialog procedure tests to see if a value has been entered in the type-in box. If the type-in box has been completed, the dialog procedure returns True so the ACCEPT trigger will be processed. If the type-in box is empty, the dialog procedure displays a message, positions the cursor on the type-in box, and returns False to deny the ACCEPT trigger and force the user to enter a value.

---

**Using the  
REFRESHCONTROL  
command**

The REFRESHCONTROL command refreshes a specified control element and implicitly issues a RESYNCCONTROL. For example, if you create a dialog box that displays a pick list of files in a user-specified directory, REFRESHCONTROL can reevaluate and redisplay the pick list based on the path that the user specifies. Figure 15-16 illustrates a dialog box with a pick list that references a user-specified directory.

Figure 15-16 Pick list displaying files in a user-specified directory



The following code uses REFRESHCONTROL to maintain the dialog box shown in Figure 15-16:

```

PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
    ; proc is called only on update trigger
    IF TagValue = "DirTag"
        ; if the ACCEPT type-in box changes
        THEN DirString = EventValue
        ; reassign variable after it changes
        REFRESHCONTROL "PickFileTag"
        ; refresh pick list using the new
        ; directory specification
        ; such as either pushbutton
        ; process update event
    ENDIF
    RETURN True
ENDPROC

DirString = "C:\\PDOX40\\SAMPLE\\"
; initial value for ACCEPT type-in box

SHOWDIALOG "Pick List in any Directory"
PROC "DialogProc"
; procedure name to execute
TRIGGER "UPDATE"
; call procedure for UPDATE trigger
@3,16 HEIGHT 14 WIDTH 45

PICKFILE @1,4 HEIGHT 6 WIDTH 35
; create a pick list
COLUMNS 2
DirString
TAG "PickFileTag"
TO FileName

LABEL @8,4
; create a label for the pick list
"~D~irectory: "
FOR "DirTag"

ACCEPT @8,18 WIDTH 21
; create an ACCEPT type-in box
"A20"
TAG "DirTag"
TO DirString

PUSHBUTTON @10,9 WIDTH 10
; create an OK push button
"~O~k"
OK
DEFAULT
VALUE "Accept"
TAG "OKTag"

```

```

        TO ButtonValue

    PUSHBUTTON @10,26 WIDTH 10           ; create a CANCEL push button
        "~C~ancel"
        CANCEL
        VALUE "Cancel"
        TAG "CancelTag"
        TO ButtonValue

ENDDIALOG

```

In this dialog box, the default path for the pick list is established by the initial value of the variable *DirString*. The user can specify a new path for the pick list by entering a value in the ACCEPT type-in box. After the value of the ACCEPT control changes, pressing the *Tab* key or clicking to navigate causes an UPDATE trigger to be issued and the *DialogProc* procedure to be called. In the *DialogProc*, the REFRESHCONTROL command redisplay the pick list based on the user's path specification.

The REFRESHCONTROL command has a different effect for each control element. Do not confuse the REFRESHCONTROL command with the RESYNCCONTROL command. The effects of both commands are compared and summarized in Table 15-3.

---

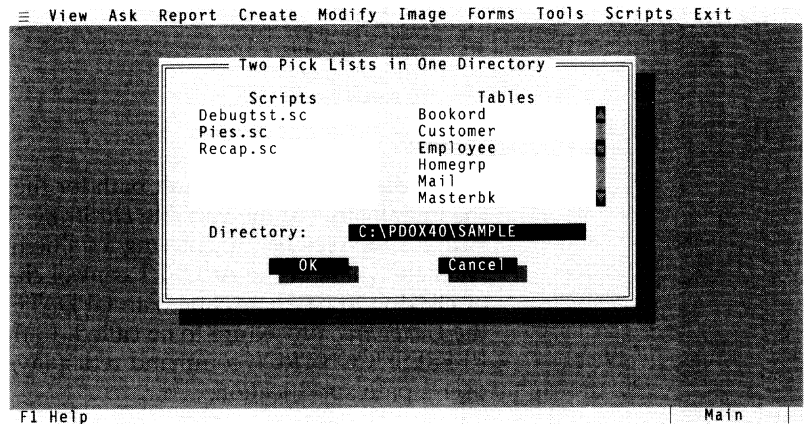
### **Using the REFRESHDIALOG command**

The REFRESHDIALOG command refreshes every control element within a dialog box and implicitly issues a RESYNCDIALOG (and therefore causes a REPAINTDIALOG). Because REFRESHCONTROL does not result in a REPAINTDIALOG, the REFRESHDIALOG command is slightly different than issuing a REFRESHCONTROL for every control element.

If you create a dialog box with two pick lists that reference a user-specified directory, you can use REFRESHDIALOG to reevaluate and redisplay both pick lists based on the path that the user specifies, rather than issuing a REFRESHCONTROL for every control element. However, you could gain a performance advantage by simply issuing multiple REFRESHCONTROL commands and avoiding the REPAINTDIALOG that REFRESHDIALOG implies.

For example, Figure 15-17 illustrates a dialog box with two pick lists that reference the same user-supplied directory.

Figure 15-17 Two pick lists in a user-specified directory



The following code uses REFRESHDIALOG to maintain the dialog box shown in Figure 15-17:

```

PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
    ; proc is called only on update trigger

    IF TagValue = "DirTag"
        THEN DirString = EventValue
        REFRESHDIALOG
    ENDIF
    RETURN True
    ; if the ACCEPT type-in box changes
    ; reassign variable after it changes
    ; refresh both pick lists using
    ; the new directory specification
    ; process the event whenever an
    ; update trigger is generated

ENDPROC

DirString = "C:\\PDOX40\\SAMPLE\\" ; initial value for ACCEPT type-in box

SHOWDIALOG "Two Pick Lists in One Directory"
PROC "DialogProc"
TRIGGER "UPDATE"
@3,15 HEIGHT 15 WIDTH 48
@1,8 ?? "Scripts Tables"

PICKFILE @2,2 HEIGHT 6 WIDTH 19 ; create a pick list
DirString + "*.SC" ; display scripts in the pick list
TAG "PickFileTag"
TO FileName

PICKTABLE @2,24 HEIGHT 6 WIDTH 19 ; create a pick list of tables
DirString
TAG "PickTableTag"
TO TableName

LABEL @9,3
"-D-irectory: " ; create a label for the pick list
FOR "DirTag"

ACCEPT @9,18 WIDTH 24 ; create an ACCEPT type-in box
"A18"
TAG "DirTag"
TO DirString
    
```



```

PUSHBUTTON @11,9 WIDTH 10           ; create an OK push button
  "~O-K"
  OK
  DEFAULT
  VALUE "Accept"
  TAG "OKTag"
  TO ButtonValue

PUSHBUTTON @11,26 WIDTH 10          ; create a CANCEL push button
  "~C~ancel"
  CANCEL
  VALUE "Cancel"
  TAG "CancelTag"
  TO ButtonValue

```

```
ENDDIALOG
```

In this dialog box, the user can specify a directory for a pick list displays scripts and a pick list displaying tables. As in the REFRESHCONTROL example, pressing the *Enter* key causes an UPDATE trigger to be issued and the *DialogProc* procedure to be called. In the *DialogProc*, the SELECTCONTROL command keeps focus on the ACCEPT control and the REFRESHDIALOG command redisplay both pick lists based on the user's path specification.

---

### **Using the ACCEPTDIALOG command**

The ACCEPTDIALOG command accepts the current dialog box. ACCEPTDIALOG is useful in situations where you do not want to use an OK push button and in situations where you want to accept the dialog box without forcing the user to press the OK button. For example, you frequently want double-clicking a pick list item (or pressing the *Space* key on a selected item) to accept a dialog box. The following example traps for the SELECT trigger that is generated in this situation and uses the dialog procedure to accept the dialog box:

```

PROC SelectProc(TriggerType, TagValue, EventValue, ElementValue)
  ACCEPTDIALOG
  VIEW TableDir + TableName
ENDPROC

TableDir = "C:\PDOX40\SAMPLE\"

SHOWDIALOG "List of Tables"           ; begin dialog box definition
PROC "SelectProc"                     ; name of procedure to execute
TRIGGER "SELECT" , "ACCEPT"          ; call procedure for SELECT and ACCEPT
@4,24 HEIGHT 15 WIDTH 29             ; location and dimensions of dialog box

PICKTABLE @1,6                         ; create a pick list
  HEIGHT 8 WIDTH 14
  TableDir
  TAG "TableTag"
  TO TableName

PUSHBUTTON @11,2 WIDTH 10             ; create an OK push button
  "OK"
  OK
  DEFAULT
  VALUE "Accept"
  TAG "OKTag"
  TO ButtonValue

```

```

PUSHBUTTON @11,14 WIDTH 10      ; create a CANCEL push button
  "Cancel"
  CANCEL
  VALUE "Cancel"
  TAG "CancelTag"
  TO ButtonValue

```

```

ENDDIALOG

```

In the above example, the dialog procedure *SelectProc* is called for both SELECT and ACCEPT triggers, because *SelectProc* also views the table that the customer selects.

---

### Using the **CANCEL DIALOG** command

The CANCEL DIALOG command cancels the current dialog box. Control variables used within the dialog box are not assigned if a CANCEL DIALOG is issued; control variables are restored to the value they had when the dialog box was opened. CANCEL DIALOG is useful in situations where you do not want to use an CANCEL push button and in situations where you want to cancel the dialog box automatically. For example, you could let the user cancel a dialog by pressing *Ctrl-Q* as follows:

```

PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
  IF TriggerType = "EVENT"      ; if proc called for Ctrl-Q
    THEN CANCEL DIALOG         ; then cancel dialog
    ELSE ACCEPT DIALOG         ; otherwise accept dialog
  ENDIF
ENDPROC

TableDir = "C:\PDOX40\SAMPLE\"

SHOW DIALOG "List of Tables"    ; begin definition
  PROC "DialogProc"            ; name of procedure to execute
  KEY 17                       ; trap for Ctrl-Q to cancel
  @4,24 HEIGHT 15 WIDTH 29     ; location and dimension of box

  PICK TABLE @1,6             ; create a pick list
    HEIGHT 8 WIDTH 14
    TableDir
    TAG "TableTag"
    TO TableName

  PUSHBUTTON @11,2 WIDTH 10    ; create an OK push button
    "OK"
    OK
    DEFAULT
    VALUE "Accept"
    TAG "OKTag"
    TO ButtonValue

  PUSHBUTTON @11,14 WIDTH 10   ; create a CANCEL push button
    "Cancel"
    CANCEL
    VALUE "Cancel"
    TAG "CancelTag"
    TO ButtonValue
END DIALOG

IF Retval                      ; Retval is True if dialog was accepted
  THEN VIEW TableDir + TableName ; if Retval = True view table

```

```

ELSE MESSAGE "Cancelling dialog..." ; otherwise display message
      SLEEP 1000
ENDIF

```

---

## Using the NEWDIALOGSPEC command

The NEWDIALOGSPEC command is used to change the event list that the dialog procedure in the active SHOWDIALOG command is called for. NEWDIALOGSPEC is typically used within a SWITCH...ENDSWITCH statement that can respond differently to the various events that the dialog procedure is called for.

Because the user can interact with most control elements in a dialog box using either the keyboard or the mouse, you frequently trap for triggers in a SHOWDIALOG statement; as described in Chapter 13, "Handling events," triggers are independent of the physical device that initiates them. However, the ACCEPT type-in box differs from other control elements because it is intended to receive exclusively keyboard input from the user. You may want to trap for keyboard events when the user is *within* an ACCEPT type-in box to react to specific keys, and trap for triggers in the rest of the dialog where the type of physical interaction is not important.

For example, you could use a bank of radio buttons and an ACCEPT type-in box in a "Method of Payment" dialog box. The radio buttons could indicate payment alternatives such as "Visa" and "MasterCard"; the type-in box could receive the credit card number. You could trap for UPDATE triggers in the general dialog box interaction to see where the user is moving and key events in the ACCEPT statement to perform syntax checking of the credit card number.

To illustrate one possible use of NEWDIALOGSPEC, the following code creates a dialog box with two type-in boxes; the second type-in box converts all upper case keypresses to lower case keys and enters them:

```

PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
  IF TagValue = "LowerTag"
    THEN SWITCH
      CASE TriggerType = "ARRIVE" : ; ARRIVE trigger changes
        NEWDIALOGSPEC TRIGGER "DEPART" KEY "ALL" ; the event list
      CASE TriggerType = "DEPART" : ; DEPART restores
        NEWDIALOGSPEC TRIGGER "ARRIVE" ; original event list
      CASE TriggerType = "EVENT" : ; keypress processed
        IF EventValue["KEYCODE"] <= 90 AND ; if keycode between 90
          EventValue["KEYCODE"] >= 65 ; and 65 (upper case)
          THEN ; make it lower case
            EventValue["KEYCODE"] = ASC(LOWER(CHR(EventValue["KEYCODE"])))
            RELEASE VARS EventValue["SCANCODE"] ; release scancode so
                                                    ; it does not conflict
                                                    ; with keycode
        ENDIF
      ENDSWITCH
    ENDIF
    RETURN True ; execute the event
  ENDPROC

```

```

SHOWDIALOG "Two type-in boxes"
  PROC "DialogProc" TRIGGER "ARRIVE"
    @4,22 HEIGHT 14 WIDTH 35

    @1,5 ?? "This box ignores case:"
    @5,5 ?? "This box makes text"
    @6,5 ?? "appear as lower case:"

    ACCEPT @2,5 WIDTH 23           ; create an ACCEPT type-in box
      "A21"
      TAG "FirstTag"
      TO FirstValue

    ACCEPT @7,5 WIDTH 23           ; create an ACCEPT type-in box
      "A21"
      TAG "LowerTag"
      TO LowerValue

    PUSHBUTTON @10,4 WIDTH 10      ; create an OK push button
      "OK"
      OK
      DEFAULT
      VALUE "Accept"
      TAG "OKTag"
      TO ButtonValue

    PUSHBUTTON @10,19 WIDTH 10     ; create a CANCEL push button
      "Cancel"
      CANCEL
      VALUE "Cancel"
      TAG "CancelTag"
      TO ButtonValue

  ENDDIALOG

```

In the above code sample, the dialog procedure is originally called for ARRIVE triggers that occur within the dialog box. If the ARRIVE trigger is caused by the user arriving in the second type-in box, the dialog procedure is passed *LowerTag* as the *TagValue*, and the SWITCH...ENDSWITCH statement evaluates the event. In this situation the CASE *TriggerType* is ARRIVE and the resulting NEWDIALOGSPEC statement changes the event list of the SHOWDIALOG command; the dialog procedure is now called for the DEPART trigger and all key events.

As the user types characters into the second type-in box, the dialog procedure is called repeatedly; because the CASE *TriggerType* is EVENT for all key events, the characters are converted into lower case. When the user leaves the type-in box, the dialog procedure is called by the DEPART trigger; because the CASE *TriggerType* is DEPART, the NEWDIALOGSPEC command restores the original event list.

You can, of course, also use a PICTURE clause with the ACCEPT statement to specify case; however, by trapping for triggers and using NEWDIALOGSPEC, you can perform extensive text manipulation.

---

**Using the  
CONTROLVALUE() function**

The CONTROLVALUE() function determines the value of any control element. CONTROLVALUE() is most useful when the value of the control element is different from the value of the control element variable.

The values of a control element and its variable are usually the same; these values differ only in special situations. For example, the value that a user enters in an accept control is not passed to the accept control variable until an UPDATE trigger occurs.

CONTROLVALUE() can be used to determine the value of any control element; however, CONTROLVALUE() is most useful when you are trapping for key events in an accept control element and you want to know the current user-entered value.

The complete syntax of the CONTROLVALUE() function follows:

**CONTROLVALUE ( Tag [ , Label ] )**

To determine the value of any control except a check box, use CONTROLVALUE and specify the *Tag*. To determine the value of a check box, specify its *Tag* and its *Label*.

For example, the following script uses CONTROLVALUE() to determine the value that a user enters in a type-in box and supplies that value to a LOCATE command. This script assumes that the *Orgchart* table already exists and contains Dept and Help fields.

```
PROC DialogProc(TriggerType, TagValue, EventValue, ElementValue)
  IF TagValue = "AcceptTag" AND                ; if current control is
    NOT ISBLANK(CONTROLVALUE(TagValue))        ; the type-in box, and
    THEN                                       ; it's not blank
    THEN                                       ; find first record with
    LOCATE PATTERN CONTROLVALUE(TagValue) + ". ." ; the type-in box value
    IF Retval                                ; if LOCATE is successful
    THEN WINDOW CREATE FLOATING @4,25 HEIGHT 5 WIDTH 42 TO HelpWin
      @1,1 ?? [Help]                          ; display text in a window
      MESSAGE "Press any key when done"
      GETEVENT KEY "ALL" TO Retval            ; eat the next key event
      WINDOW CLOSE
      MESSAGE ""
    ELSE MESSAGE "No help available."          ; locate failed, inform user
      SLEEP 1500
      MESSAGE ""
    ENDIF
    RETURN False                               ; don't let Paradox process F1 key
  ELSE MESSAGE "No help for blank fields"
    SLEEP 1500
    MESSAGE ""
    RETURN True                                ; let Paradox process the F1 key
  ENDIF
ENDPROC

VIEW "Orgchart"                               ; table with [Dept] and [Help] fields
MOVETO FIELD "Dept"

SHOWDIALOG "Data-sensitive help"
PROC "DialogProc"                             ; procedure name to execute
```

```

KEY "F1" ; call procedure when F1 key pressed
@6,8 HEIGHT 11 WIDTH 40

@1,0 ?? FORMAT("W38,AC","Enter all or part of Dept name")
@2,0 ?? FORMAT("W38,AC","Press [F1] for help")

ACCEPT @4,9 WIDTH 20 ; create an ACCEPT type-in box
"A12"
TAG "AcceptTag"
TO SearchString

PUSHBUTTON @7,5 WIDTH 10 ; create an OK push button
"~O~K"
OK
DEFAULT
VALUE "Accept"
TAG "OKTag"
TO ButtonValue

PUSHBUTTON @7,22 WIDTH 10 ; create a CANCEL push button
"~C~ancel"
CANCEL
VALUE "Cancel"
TAG "CancelTag"
TO ButtonValue

```

ENDDIALOG

In the above script, the *Orgchart* table is placed on the workspace and the workspace cursor is positioned in the Department field. The dialog box prompts the user to enter a value in the type-in box. If the user presses *F1* for help and the cursor is positioned on the accept control, the `CONTROLVALUE()` function supplies the current value of the accept control element to the `LOCATE` command.

The value of the *FileFilter* variable is not useful here because the value entered in the accept control has not yet been bound to the accept control variable. The accept control variable is bound after an `UPDATE` or `SELECT` trigger has occurred.

# Using the WAIT command

Paradox lets you interact with the user in powerful and flexible ways. The WAIT command lets you leave a user in a multi-table form, table, record, or field, and wait until an event that you want to act upon occurs. A WAIT statement waits for events to happen and makes your application “event-driven.”

This chapter covers:

- waiting for specified keypresses with an UNTIL *KeyCodeList*
- trapping for any Paradox event with an *EventList*
- using the WAIT command while a user interacts with a field, record, table, or multi-table form
- table navigation during a WAIT

---

## Interacting with the user

The WAIT command lets the user view or edit one or more tables (in either table or form view) that are already on the workspace. When the WAIT command is executed, the user can interact with the table, either viewing it or editing it (if you’ve put the workspace in Edit or CoEdit mode). Follow the WAIT command with the FIELD, RECORD, TABLE, or WORKSPACE keyword to limit the interaction to the current field, record, table, or the entire workspace, respectively.

A WAIT statement can be as simple or complicated as you deem necessary. The simplest kind of WAIT tests for a specific key or for any one of a set of keys. For example, you can construct a WAIT FIELD that waits for the user to press the *Enter* or *Tab* keys to move between fields. A more sophisticated WAIT tests for events or triggers that indicate what action the user is taking.

During a WAIT interaction, all validity checks and image settings remain in effect. For example, you can use any of the following features that are available in Paradox:

- ValCheck | Picture to set up a picture for the field that entered values must match. For example, the picture ###-##-#### restricts values to numbers in Social Security format—and even fills in the hyphens automatically.
- ValCheck | TableLookup to make sure that entered values match the first field in another table. In addition, the AllCorrespondingFields option fills in all other fields with the same name as fields in the lookup table, while the HelpAndFill option lets users press Help F1 to browse through the lookup table or use Zoom Ctrl-Z to find the record they want.

For example, in an order entry form, setting up a validity check using the TableLookup | AllCorrespondingFields | HelpAndFill option in the Cust ID field would ensure that the value matched a Cust ID in the *Customer* table and fill in the customer's name and address automatically. The user could press Help F1 to browse through *Customer* and find the correct record.

For more information about these possibilities, see “Validity checks” in Chapter 11 of the *User's Guide*.

You can use the WAIT command repeatedly in the course of a script. For example, an order entry script might contain a loop in which the user is prompted to fill out a form and then press Do\_It! F2. Then, on the workspace behind the canvas, the script could post parts of the order to the *Customer* and *Billing* tables. Finally, control could return to the top of the loop and another blank order form would be displayed.

Chapter 17 discusses special considerations you should be aware of when you want the user to interact with multi-table forms through a WAIT command.

---

## Two forms of the WAIT command

Paradox lets you use both a keypress-driven WAIT command and an event-driven WAIT that gives you access to the full range of Paradox events and triggers. The keypress-driven WAIT was originally designed for versions of Paradox prior to 4.0, although it is still useful when you only want to respond to keypresses during a WAIT interaction. The event-driven WAIT, however, gives you access to the extensive event handling capabilities of Paradox.



The keypress-driven WAIT is available in compatible mode and standard mode; the event-driven WAIT is only available in standard mode.

---

## Keypress-driven WAIT

When you use the keypress-driven WAIT, your PAL script is suspended and a user can interact with the workspace until pressing a key that you define. When that key is pressed, the user is done working with the table and your script resumes.

Following is the complete syntax of the keypress-driven WAIT:

```
WAIT { FIELD | RECORD | TABLE | WORKSPACE }  
      [ PROMPT ExpressionList ]  
      [ MESSAGE Expression ]  
UNTIL KeycodeList
```

The keypress-driven WAIT waits for keypresses specified in the UNTIL list. All keypresses not listed in the UNTIL list of the WAIT statement are passed on to Paradox for processing.

The FIELD, RECORD, TABLE, and WORKSPACE keywords are used to limit the extent of the WAIT interaction. For example, a WAIT FIELD permits the user to interact with the current field only, issuing a beep if the user attempts to leave that field. Similarly, WAIT RECORD, TABLE, or WORKSPACE permit an interaction with the current record, current table, or the entire workspace. The interaction continues until a keypress specified in the UNTIL *KeycodeList* occurs.

You can use the PROMPT or MESSAGE statements within the WAIT to display information to the user, such as which key to press when finished.

Example 16-1 shows you how to use the WAIT command to permit a simple table interaction.

---

### Example 16-1 Basic table interaction in a WAIT

The following procedure uses a WAIT interaction to let the user browse through a table and pick a particular record of interest by pressing Do\_It! F2:

```
PROC FindRecord(Tabname)  
  
    VIEW Tabname                ; place the table on the workspace  
    WAIT TABLE                 ; let the user browse the table  
    PROMPT "Move the cursor to the desired record, then press [F2].",  
          "Press [Esc] to quit"  
    UNTIL "F2", "Esc"  
  
    IF Retval = "Esc"           ; user has not selected a record  
    THEN RETURN False  
    ELSE COPYTOARRAY SelectRec ; copy contents of record
```

```

; to selectrec array
ENDIF
ENDPROC

```

If your purpose is to let a user modify one or more tables, you will find it useful to place a keypress-driven WAIT interaction inside a WHILE loop. Place the necessary tables on the workspace, issue an EDITKEY or COEDITKEY to permit the user to modify them, then start a WHILE loop. Inside the WHILE loop, open a WAIT statement and let the user edit the table. Example 16-2 shows you how to use the WAIT command inside a WHILE loop.

### Example 16-2 Using WAIT in a WHILE loop

This example lets the user coedit a table within a WHILE loop. When the user presses Do\_It! F2, the WAIT ends (but not the WHILE or the coedit). Your script posts the record, then loops to the top of the WHILE and starts the WAIT again. Effectively, this lets you monitor the user and selectively respond to particular events without ending the coediting or viewing session.

```

VIEW "Orders"           ; place the Orders table on the workspace
END                     ; go to the last record
PICKFORM 2              ; display using custom form F2
COEDITKEY               ; enter CoEdit mode
WHILE(True)             ; begin the WHILE loop
  PGDN                   ; open a new record at the end of the table
  WAIT RECORD            ; let the user directly interact with
                        ; the table, but only one record at a time
  PROMPT "Press F2 when done with record, Esc to quit."
  UNTIL "F2", "Esc"     ; F2 ends the WAIT but not the WHILE loop
  IF Retval = "Esc"     ; Esc ends the WAIT and the WHILE loop
    THEN QUITLOOP
  ENDF
  .                       ; insert additional code here to post the record,
  .                       ; examine and handle key violations, and so forth
  .                       ; on a record-by-record basis
ENDWHILE
DO_IT!                  ; end the coedit

```

Do not use a WHILE loop with an event-driven WAIT; you control the duration of the interaction with the WAIT procedure. The next section explains the event-driven WAIT in detail.

---

## Event-driven WAIT

When you use the event-driven WAIT, your PAL script is suspended and a user can interact with the workspace until an event or trigger that you specify in the *EventList* occurs. Following is the complete syntax of the event-driven WAIT:

```

WAIT { FIELD | RECORD | TABLE | WORKSPACE }
PROC WaitProc EventList
ENDWAIT

```

Like the keypress-driven WAIT, the keywords FIELD, RECORD, TABLE, and WORKSPACE in the event-driven WAIT let you wait while a user interacts with either a field, record, table, or the workspace.

Unlike the keypress-driven WAIT, the PROMPT and MESSAGE statements are not legal within the event-driven WAIT...ENDWAIT. Specify PROMPT, MESSAGE, or both before entering the WAIT, or use the *WaitProc* procedure to set PROMPT and MESSAGE. The PROMPT is cleared automatically after the ENDWAIT statement. Clear the MESSAGE after the ENDWAIT by issuing a MESSAGE statement followed by an empty string.

The *WaitProc* procedure is similar to the *DialogProc* procedure that is used in the SHOWDIALOG command. Like the *DialogProc*, the *WaitProc* procedure is called when an event on the *EventList* occurs; you can use the *WaitProc* procedure to examine the event and take the appropriate action. The *EventList* for the WAIT command includes any or none of the standard Paradox events listed in Appendix J of the *PAL Reference* and described in detail in Chapter 13.

When you enter the WAIT, global echo is set to normal regardless of the previous echo setting; global echo is restored to its previous setting when you leave the WAIT. When the *WaitProc* procedure is called, echo is set to the same setting it had prior to entering the WAIT; when the *WaitProc* procedure returns, echo is normal again in the WAIT.

Because changing the global echo state repeatedly can be time consuming, you may want to set global echo to normal immediately before you enter the WAIT and set global echo to off when you leave the WAIT. Setting the echo state in this manner will make event processing with the *WaitProc* procedure much faster because Paradox won't have to toggle between ECHO NORMAL and ECHO OFF every time the *WaitProc* procedure is called. See Chapter 12 for further information about the ECHO command.

---

## Using the WAIT procedure

The WAIT procedure can be called for any mouse, key, message, or idle event, or any of the triggers listed in Table 13-9. The following discussion explains how to take advantage of event-driven programming during a WAIT interaction. If you are uncertain about how to use events or triggers, you should review Chapter 13, "Handling events."

The WAIT procedure *WaitProc* takes three arguments to help you determine the exact situation in which it was called. You can name these arguments differently when you use them; for the purposes of this manual, we have named them as follows:

*WaitProc* (*TriggerType*, *EventRecord*, *CycleNumber*)

The *TriggerType* and *EventRecord* arguments are assigned values by the event that *WaitProc* is called for; *CycleNumber* is assigned a unique random value that stays the same throughout a trigger cycle by Paradox.

*TriggerType* is assigned a value by the type of trigger or event that the *WaitProc* is called for. If *WaitProc* is called for a trigger, the specific name of the trigger, such as "ARRIVEFIELD" or "ARRIVEROW", is passed to *TriggerType*. If *WaitProc* is called for a mouse, key, message, or idle event, *TriggerType* is passed the string "EVENT".

The calling trigger or event also causes the dynamic array *EventRecord* to be created. If *WaitProc* is called for a mouse, key, message, or idle event, the contents of *EventRecord* are the same as the contents of a dynamic array that GETEVENT would create for that event. If *WaitProc* is called for a trigger, the specific physical action that results in the trigger, such as a mouse click or a keypress, is represented in the dynamic array. In this case, the contents of the dynamic array are also the same as the contents of a dynamic array that GETEVENT would create for that event. See the description of the GETEVENT command in the *PAL Reference* for details about this dynamic array.

*CycleNumber* is a unique value that stays the same within a particular trigger cycle. The trigger cycle is the complete set of triggers arising from a single user action. For example, a single user action such as leaving the last field in a record by pressing → would result in the DEPARTFIELD, DEPARTROW, ARRIVEROW, and ARRIVEFIELD triggers in the order listed here. These triggers all occur within the same trigger cycle because they all arise from the same single user action.

The triggers in any trigger cycle occur in a specific sequence. Because a single user action can result in a number of triggers, PAL provides an opportunity for the developer to detect very specific parts of a trigger cycle. The sequence in which triggers occur in a WAIT interaction depends on the location of the cursor, the mode of the workspace (such as Main, Edit, CoEdit, and so forth), and the type of action the user takes.

For example, if the cursor is in the first field of a record with more than one field, pressing the → key results in the DEPARTFIELD and ARRIVEFIELD triggers in any mode. If the cursor is in the last field of the last record, pressing the → key results in the trigger sequence DEPARTFIELD, DEPARTROW, ARRIVEROW, and ARRIVEFIELD in CoEdit mode; however, the same action results in no triggers in Main mode.

The *WaitProc* procedure must return a value of 0, 1, or 2 so Paradox knows what action to take with the WAIT. If the event or trigger that

the *WaitProc* procedure is called for is a *before* event or trigger, it is still *pending* when the procedure returns. The value the *WaitProc* returns also specifies how to process this calling event. Table 16-1 summarizes the action that results from each of these values you can return from the *WaitProc* procedure.

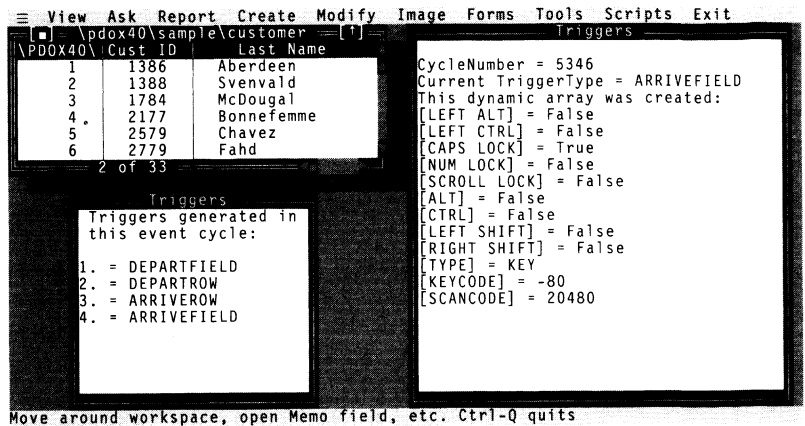
Table 16-1 WAIT procedure return values

Return value	Resulting action
0	Process the current event and proceed to the next event in the same trigger cycle
1	Deny the current event and break out of the trigger cycle but not the WAIT
2	Deny the current event and break out of both the trigger cycle and the WAIT

Notice that returning a value from the *WaitProc* procedure makes using a WHILE loop with a WAIT—the procedure described in Example 16-2 for the keypress-driven WAIT—unnecessary. If you return the value 0 from the *WaitProc* procedure, a pending event is passed on to Paradox and processed normally, and the WAIT continues with the next event in the same trigger cycle. If you return the value 1 or 2 from the *WaitProc* procedure, a pending event is *denied*; that is, it is not passed on to Paradox and processed. The value 1 ends the trigger cycle but continues the WAIT; the value 2 ends both the trigger cycle and the WAIT. See Chapter 13, “Handling events,” for complete information about trigger cycles. Table 13-9 and Appendix J of the *PAL Reference* both list the triggers available in the WAIT command.

The best way to learn about event processing within the WAIT is to trap for triggers and use the *WaitProc* procedure to display the trigger sequence for an entire trigger cycle onscreen. Figure 16-1 shows the value of the *WaitProc* procedure arguments and the trigger sequence for a typical WAIT TABLE. Example 16-3 shows how you can use the *WaitProc* procedure to display this information about the WAIT.

Figure 16-1 Trigger sequences in a WAIT



Example 16-3 Trigger sequences in a WAIT

In this example, the *ShowEvents* procedure is called for every trigger that occurs. The current cycle number—the value of the *CycleNumber* argument—is displayed on the canvas on the right side of the screen. This canvas also displays the values of the elements in the dynamic array created for the trigger and passed to *EventRecord* argument. The canvas on the lower left of the screen sequentially displays all the triggers that occur in the current cycle.

As you move through the table with the mouse and the keyboard, different sequences of triggers occur and new values are displayed on the two canvases. Figure 16-1 shows the effect of running this script.

PROC ShowEvents(TriggerType, EventRecord, CycleNumber)

```

; display all triggers for a single event cycle (CycleNumber)

SameCycle = (LastCycle = CycleNumber) ; SameCycle = true if CycleNumber
; hasn't changed
SETCANVAS TriggerListWin ; set canvas focus
IF SameCycle ; if Cycle number hasn't changed
THEN TrigNum = TrigNum + 1 ; increment number of triggers
; in the same CycleNumber
? TrigNum, ". = ", TriggerType ; display the trigger
ELSE @2,0 CLEAR EOS ; if CycleNumber has changed
LastCycle = CycleNumber ; store the current CycleNumber
TrigNum = 1 ; reset trigger number
? TrigNum, ". = ", TriggerType ; display type of trigger
ENDIF
SETCANVAS TriggerWin ; set canvas focus
CLEAR
@0,0

; display the formal parameters that describe the current trigger
? "CycleNumber = ", CycleNumber ; display formal parameters of
? "Current TriggerType = ", TriggerType ; wait procedure

```

```

? "This dynamic array was created:"
FOREACH Element IN EventRecord           ; display the dynamic array that
                                         ; describes the trigger event
    ? "[", Element, "]" = ", EventRecord[Element]
ENDFOREACH
ECHO NORMAL
SLEEP 1000
IF EventRecord["TYPE"] = "KEY"
    AND EventRecord["KEYCODE"] = 17     ; if user presses Ctrl-Q
    THEN RETURN 2                       ; break wait session
ENDIF
RETURN 0                                 ; process the trigger event

ENDPROC

LastCycle = BLANKNUM()                  ; initialize tracking variables
TrigNum = 1

DYNARRAY WinAtts[]                    ; set up dynamic array for
WinAtts["TITLE"] = "Triggers"          ; window attributes
WinAtts["HASSHADOW"] = False
WinAtts["MARGIN"] = 1
WinAtts["CANCLOSE"] = False

VIEW "Customer"
WINDOW HANDLE IMAGE 1 TO CustWindow
WINDOW RESIZE CustWindow TO 9,38       ; resize Customer to small window

                                         ; create watch windows
WINDOW CREATE ATTRIBUTES WinAtts @1,40 WIDTH 39 HEIGHT 23 TO TriggerWin
WINDOW CREATE ATTRIBUTES WinAtts @11,6 WIDTH 25 HEIGHT 13 TO TriggerListWin
@0,1 ?? "Triggers generated in"        ; write to TriggerListWin window
@1,1 ?? "this event cycle: "

WINDOW SELECT CustWindow                ; make Customer current window

PROMPT "Move around workspace, open Memo field, etc. Ctrl-Q quits"
SHOWPULLDOWN                            ; hide Main Paradox menu
ENDMENU                                  ; with blank menu
WAIT WORKSPACE
    PROC "ShowEvents"                   ; call WatchTriggers() procedure
        TRIGGER "ALL"                   ; on all possible triggers
        KEY 17                          ; and Ctrl-Q
    ENDWAIT
CLEARPULLDOWN                            ; remove SHOWPULLDOWN menu
RETURN "End of script, back in Paradox now..."

```

Notice that all the windows displayed onscreen are available; moving between windows causes different sequences of triggers. You can even move to the last field of the Customer table—a memo field—and enter field view. Because entering field view in a memo field opens an Editor session and a window, an ARRIVEWINDOW trigger is issued here. You can trap for the ARRIVEWINDOW trigger in your application and then use ISFIELDVIEW() to determine when the user moves into field view in a memo field.

---

## Using a SHOWPULLDOWN menu

In Example 16-3, a SHOWPULLDOWN...ENDMENU statement is issued immediately before entering the WAIT. As described in Chapter 14, using SHOWPULLDOWN in this manner installs an empty SHOWPULLDOWN menu that conceals the Paradox Main menu.

If you do not use a SHOWPULLDOWN menu, the Paradox Main menu is still visible when you enter the WAIT; however, it is not available to the user. You will frequently want to use a SHOWPULLDOWN menu during the WAIT either to hide the Paradox Main menu or to make your own menu choices available to the user.

If the user opens an Editor session during the WAIT, either by entering field view in a memo field, using MiniEdit *Alt-E*, or by using any other technique, the Paradox Main menu is replaced with the Editor menu unless you use a SHOWPULLDOWN menu. Unlike the Main menu, the Editor menu is available to the user during the WAIT to allow access to the editing commands. To have more control over the user during an Editor session within a WAIT, you may also want to use a SHOWPULLDOWN menu to provide a subset of the Editor functionality. See “Changing the SHOWPULLDOWN menu with NEWWAITSPEC” later in this chapter for information about changing the SHOWPULLDOWN menu to correspond to different situations in the workspace during a WAIT interaction.

---

## Setting Retval from the WAIT procedure

Because the WAIT is suspended while the *WaitProc* is executing, many actions that occur during the *WaitProc* may cause the WAIT to end when the *WaitProc* returns. If the WAIT is ended when the *WaitProc* procedure returns, *Retval* is set to indicate what action terminated the WAIT. For example, if the last image on the workspace is closed during the *WaitProc*, the WAIT is ended and *Retval* is set to 2004 when the *WaitProc* returns. Table 16-2 describes the values that the *WaitProc* procedure can assign to *Retval*.

Table 16-2 Values assigned to Retval by WaitProc procedure

---

Retval value	Assigned by
2002	Returning 2 from the WaitProc procedure (normal termination)
2003	Moving to a different field, record, or table in the WAIT procedure and not returning to the original field, record, or table before returning to the WAIT during a WAIT FIELD, WAIT RECORD, or WAIT TABLE
2004	Closing the last image on the workspace during a WAIT
2005	Returning a value other than 0, 1, or 2 from the WaitProc procedure

---



---

## Table navigation during a WAIT

You do not have to do anything special to let a user move between tables during a WAIT. If you place two tables on the workspace and then issue a WAIT WORKSPACE, Paradox will let the user move between tables with the mouse or by using keys such as *F3*, *F4*, and *Ctrl-F4*.

Example 16-4 shows you how to let Paradox move between tables automatically when the user presses *F3* and *F4*.

---

### Example 16-4 Basic table navigation during a WAIT

You do not have to trap for navigation keys such as *F3*, *F4*, and *Ctrl-F4* during a WAIT WORKSPACE; Paradox will handle them for you automatically, as shown in this example.

```
PROC WaitProc(TriggerType, EventRecord, CycleNumber)
  IF EventRecord["KEYCODE"] = ASC("F2")      ; if F2 key is pressed
    THEN RETURN 2                             ; end WAIT session
  ENDF
ENDPROC

PROC SwapTables(Table1, Table2)
  VIEW Table1                                 ; view both tables maximized
  VIEW Table2

  PROMPT "Press [F3] or [F4] to switch tables, [F2] to quit"
  SHOWPULLDOWN                               ; hide Main Paradox menu
  ENDMENU                                     ; with blank menu
  WAIT WORKSPACE
    PROC "WaitProc"                           ; call this procedure
    KEY "F2"                                   ; when F2 is pressed
  ENDWAIT

  CLEARPULLDOWN                              ; remove SHOWPULLDOWN menu
  CLEARALL                                    ; clear images when WAIT ends
ENDPROC

SWAPTABLES("Employee", "RepPerf")
```

When the user presses one of the navigation keys in this example, the expected table is selected.

In some situations you will want more control over how the user moves between tables. Example 16-5 shows you how to use the *WaitProc* procedure to move between tables when the user presses *F3* and *F4*.

---

### Example 16-5 Table navigation with the WAIT procedure

In this example, the WAIT WORKSPACE traps for *F3* and *F4*. The *WaitProc* procedure then explicitly handles the navigation and returns 1 to the WAIT to deny the pending event.

```
PROC WaitProc(TriggerType, EventRecord, CycleNumber)
  IF EventRecord["KEYCODE"] = ASC("F2")      ; if F2 key is pressed
    THEN RETURN 2                             ; end WAIT session
  ELSE                                        ; must have pressed F3 or F4
    OutaSite = WINDOWAT(500,500)             ; store handle for off
                                              ; screen window
  ENDIF
ENDPROC
```

```

        WINDOW MOVE GETWINDOW() TO 500,500 ; move visible window off screen
        WINDOW MOVE OutaSite TO 1,0 ; move off screen window on screen
        WINDOW SELECT OutaSite ; make on screen window
                                ; the current window
        RETURN 1 ; end trigger cycle but
                ; not the WAIT session
    ENDIF
ENDPROC

PROC SwapTables(Table1, Table2)
    VIEW Table1 ; view both tables maximized
    Handle1 = GETWINDOW()
    WINMAX
    VIEW Table2
    Handle2 = GETWINDOW()
    WINMAX

    WINDOW MOVE Handle2 TO 500,500 ; move Table2 window off screen
    WINDOW SELECT Handle1

    PROMPT "Press [F3] or [F4] to switch tables, [F2] to quit"
    SHOWPULLDOWN ; hide Main Paradox menu
    ENDMENU ; with blank menu
    WAIT WORKSPACE
        PROC "WaitProc" ; call this procedure
        KEY "F2", "F3", "F4" ; call proc when these keys
                                ; are pressed
    ENDWAIT

    CLEARPULLDOWN ; remove SHOWPULLDOWN menu
    CLEARALL ; clear images when WAIT ends
ENDPROC

SWAPTABLES("Employee", "RepPerf")

```

In the above example, *Table1* and *Table2* are placed on the workspace and maximized; *Table2* is moved to a location where it is not visible. After the WAIT WORKSPACE is issued, the user can move between the tables by pressing **F3** and **F4**. Instead of letting Paradox handle these keys, the WAIT procedure is called and the visible table is moved out of visible range and the other table is placed onscreen. You can handle keys such as **F3** and **F4** yourself when you want more control over the default behavior of Paradox.

Notice how you can call the *WaitProc* procedure, move to a different table, and return to the WAIT on a different table when you use a WAIT WORKSPACE. If you return from the *WaitProc* procedure on a different table when you use a WAIT FIELD, WAIT RECORD, or WAIT TABLE, the WAIT will terminate and *Retval* will be set to 2003, as described in Table 16-2.

---

### **WAIT and multi-table forms**

When using multi-table forms in an application, it's important to remember that each of the separate forms embedded in the multi-table master actually represents a separate table; this is the case whether the detail forms are linked or unlinked.

Because the multi-table form actually represents more than one table, you should use a WAIT WORKSPACE to allow access to data in all the tables. In versions of Paradox prior to 4.0, allowing access to another table required you to end the first WAIT, use MOVETO, UPIMAGE, or DOWNIMAGE to move to another table, and then go

into a WAIT again. In Paradox 4.0, you simply use the WAIT WORKSPACE command to wait on every object on the workspace. Example 16-6 demonstrates how you can use WAIT with a multi-table form.

### Example 16-6 A simple multi-table WAIT procedure

Suppose you want to use a multi-table form in an application to enable users to browse through related data in several tables.

The *MultiWait* procedure opens the master table *TableName* and an associated multi-table form and lets users view data through the form as if they were using it interactively on the Paradox workspace.

```
PROC WaitProc(TriggerType, EventRecord, CycleNumber)
    RETURN 2
ENDPROC

PROC MultiWait(TableName, FormName)
    VIEW TableName
    PICKFORM FormName
    WINMAX

    PROMPT "Press [F3] for previous table, [F4] next table, [F2] to end"
    SHOWPULLDOWN
    ENDMENU
    WAIT WORKSPACE
        PROC "WaitProc"
        KEY "F2"
    ENDWAIT

    CLEARPULLDOWN
    CLEARIMAGE

ENDPROC

MultiWait("Customer", 3)
```

In the above example, the master table is explicitly placed on the workspace with the VIEW command; it is not necessary to explicitly open the detail table. The WAIT command here traps for **F2** and simply uses the *WaitProc* procedure to terminate the WAIT. In this example, Paradox handles the **F3** and **F4** keys automatically for you, letting the user move between the master table and the detail table.

Example 16-6 lets Paradox automatically handle the table navigation on a multi-table form. As mentioned previously, you can also use the *WaitProc* procedure to navigate during a WAIT. When you use the *WaitProc* procedure to move from a multi-table form to another table, you must first toggle from form view into table view and then move to another table.

For example, if the user is interacting with a multi-table form and the *WaitProc* procedure needs to find a value in a table that is not embedded in a multi-table form, you must issue a FORMKEY to

switch to table view and then move to the other table. After you find the value, move back to the table view of the master table and issue another FORMKEY to toggle into the multi-table form again. Chapter 17 gives complete information about working with multi-table forms and includes many other examples of the WAIT.

---

## Using lookup help

Because you have complete flexibility in constructing multi-table forms and can embed any combination of linked and unlinked details in the master, you can achieve some interesting effects. The WAIT WORKSPACE command makes it easy to use your multi-table forms within a WAIT. A simple but powerful way to take advantage of the lookup and fill-in support provided by Paradox's built-in validity checks is shown in Example 16-7.

---

### Example 16-7 Using default Paradox lookup help

---

Suppose you've created a multi-table form with two or more linked detail forms and created lookup help validity checks for some of the fields on the form.

This generalized procedure allows the user to interact with the form normally, and lets Paradox handle keys like **F1** (lookup help), **F3** (upimage), and **F4** (downimage). When **F2** is pressed, the WAIT procedure returns 2 to break out of the current event cycle and the WAIT session.

```
PROC WaitProc(TriggerType, EventRecord, CycleNumber)
                                ; this wait procedure is called only when
    RETURN 2                    ; F2 is pressed- when it is, the wait
                                ; procedure returns 2 to break the current
                                ; cycle, and the wait session
ENDPROC

PROC MultiWait(TableName, FormName)

    COEDIT TableName            ; coedit the table
    PICKFORM FormName          ; go into the multi-table form
    WINMAX                      ; maximize form window

    PROMPT "[F1] lookup help, [F3] previous table, " +
           "[F4] next table, [F2] to end"
    SHOWPULLDOWN                ; hide Main Paradox menu
    ENDMENU                     ; with blank menu
    WAIT WORKSPACE
        PROC "WaitProc"         ; designate wait procedure to call
        KEY "F2"                ; call "WaitProc" when F2 is pressed
    ENDWAIT

                                ; the following commands execute when
                                ; the wait proc returns 2
    DO_IT!                      ; end coedit
    CLEARPULLDOWN               ; the SHOWPULLDOWN menu is removed
    CLEARIMAGE                  ; remove image from workspace

ENDPROC

MultiWait("Customer", 3)      ; call the proc for form 3 of Customer
```

---

## Changing the event list during a WAIT

The NEWWAITSPEC command is used to change the event list that the *WaitProc* procedure in the active WAIT command is called for. NEWWAITSPEC is typically used within a SWITCH...ENDSWITCH statement that can respond differently to the various events that the *WaitProc* procedure is called for.

Because different types of interactions occur within different types of windows, you may want to use a different event list for each window within the same WAIT. For example, when the user is interacting with a table during a WAIT, you may want to trap for many keys and disable them. If the user goes into an Editor session, however, you may want to permit the use of certain keys again.

Example 16-8 illustrates one possible use of NEWWAITSPEC.

---

### Example 16-8 Changing the event list with NEWWAITSPEC

---

This example allows the user to modify a table through a single-table form. When the user enters field view in the memo field, the *WaitProc* procedure issues a NEWWAITSPEC to control the editing interaction. When the user is finished editing the memo field, the *WaitProc* procedure uses NEWWAITSPEC to restore the original event list.

```
PROC WaitProc(EventType, EventRecord, CycleNumber)
    ; first evaluate all possible events and triggers that call this
    ; wait procedure- assign the WaitRetval variable with a number
    ; that will be returned to the wait at the end of the procedure
    SWITCH
        CASE EventRecord["TYPE"] = "MESSAGE" AND
              EventRecord["MESSAGE"] = "CLOSE" :
            WaitRetval = 0
            ; if user closes the
            ; memo field window
            ; 0 value processes the
            ; close message normally
        CASE EventType = "ARRIVEWINDOW" :
            IF GETWINDOW() = FormWindow AND
              ISWINDOW(MemoWindow)
            THEN WINDOW SELECT MemoWindow
            WINDOW CLOSE
            WINDOW SELECT FormWindow
            ; cursor on a new window?
            ; if user moved to the form
            ; and a memo field is up
            ; make the memo field
            ; active, and close it
            ; then go back to the form
            ENDIF
            WaitRetval = 0
        CASE EventRecord["KEYCODE"] = ASC("F2") :
            IF ISFIELDVIEW()
            THEN WaitRetval = 0
            ELSE WaitRetval = 2
            ; if user pressed F2
            ; if we're in a memo field
            ; process F2 key event
            ; otherwise break wait
            ENDIF
        CASE EventRecord["KEYCODE"] = ASC("Esc") :
            CANCELEDIT
            WaitRetval = 0
            ; if user pressed Esc
            ; otherwise cancel editor
        CASE EventRecord["KEYCODE"] = -18 :
            BEEP
            WaitRetval = 1
            ; if user pressed Alt-E
    ENDSWITCH

    ; now that all of possible events and triggers have been
    ; handled, change the prompt and the wait spec according
    ; to the current window
    SWITCH
        ; if current window is the form, restore original prompt
        ; hide Main Paradox menu with blank menu
```

```

CASE GETWINDOW() = FormWindow :
    PROMPT "Make changes to table, press [F2] when done"
    NEWWAITSPEC TRIGGER "ARRIVEWINDOW"
        KEY "F2", -18

        ; if user has opened the memo field, display appropriate
        ; prompt, and add the events MESSAGE "CLOSE" and Esc to
        ; the wait spec
CASE ISFIELDVIEW() :
    MemoWindow = GETWINDOW() ; get handle for memo field
    PROMPT "Press [F2] to save changes to memo, Esc to cancel"
    NEWWAITSPEC TRIGGER "ARRIVEWINDOW"
        MESSAGE "CLOSE"
        KEY "F2", -18, "Esc"

ENDSWITCH

RETURN WaitRetval ; now return the appropriate value to the WAIT

ENDPROC

COEDIT "Customer"
WINDOW HANDLE IMAGE 1 TO CustWindow ; get handle for table view
WINDOW MOVE CustWindow TO 100,100 ; move table view off screen
PICKFORM 1
FormWindow = GETWINDOW() ; get handle for form window
WINDOW MOVE FormWindow TO 1,5 ; center form window

PROMPT "Make changes to table, press [F2] when done"
SHOWPULLDOWN ; hide Main Paradox menu
ENDMENU ; with blank menu
WAIT WORKSPACE
    PROC "WaitProc"
        TRIGGER "ARRIVEWINDOW" ; original wait spec only
        KEY "F2", -18 ; includes ARRIVEWINDOW and
        ; the F2 and Alt-E key events

ENDWAIT

DO_IT!
CLEARPULLDOWN ; remove SHOWPULLDOWN menu
CLEARALL

```

---

## Changing the SHOWPULLDOWN menu with NEWWAITSPEC

A SHOWPULLDOWN menu is frequently used in a WAIT interaction to let the user choose to close the WAIT table, open another table, print a report, access utilities, and so forth. Because the SHOWPULLDOWN menu is not modal, the user is free to interact with the WAIT table; the SHOWPULLDOWN menu is available when the user wants it. See Chapter 14 for complete information about the SHOWPULLDOWN command.

Example 16-9 illustrates a simple use of SHOWPULLDOWN to change the menu while coediting a memo field in a WAIT.

---

### Example 16-9 Changing a SHOWPULLDOWN menu during a WAIT

```

PROC WaitProc(EventType, EventRecord, CycleNumber)
    ; first evaluate all possible events and triggers that call this
    ; wait procedure--assign the WaitRetval variable with a number
    ; that will be returned to the wait at the end of the procedure
SWITCH
    CASE EventRecord["TYPE"] = "MESSAGE" AND
        EventRecord["MESSAGE"] = "MENUSELECT" : ; if user chooses a menu

```

```

SWITCH
  CASE EventRecord["MENUTAG"] = "Save" : ; if Save menu selected
    DO IT! ; save changes to memo
    ; field, and close window
    WaitRetval = 1 ; 1 value denies the event
    ; if Cancel menu selected
  CASE EventRecord["MENUTAG"] = "Cancel" :
    CANCELEDIT ; cancel changes to memo
    WaitRetval = 1 ; 1 value denies the event
  OTHERWISE : ; if any other menu is
    ; selected
    WaitRetval = 0 ; 0 value processes the
    ; selection normally
ENDSWITCH

CASE EventRecord["TYPE"] = "MESSAGE" AND ; if user user closes the
EventRecord["MESSAGE"] = "CLOSE" : ; memo field window
WaitRetval = 0 ; 0 value processes the
; close message normally
CASE EventType = "ARRIVEWINDOW" : ; if user moved to the form
IF GETWINDOW() = FormWindow AND ; if user moved to the form
ISWINDOW(MemoWindow) ; and a memo field is up
THEN WINDOW SELECT MemoWindow ; make the memo field
WINDOW CLOSE ; active, and close it
WINDOW SELECT FormWindow ; then go back to the form
ENDIF
WaitRetval= 0
CASE EventRecord["KEYCODE"] = ASC("F2") : ; if user pressed F2
IF ISFIELDVIEW() ; if we're in a memo field
THEN WaitRetval = 0 ; process F2 key event
ELSE WaitRetval = 2 ; otherwise break wait
ENDIF
CASE EventRecord["KEYCODE"] = ASC("Esc") : ; if user pressed Esc
CANCELEDIT ; otherwise cancel editor
WaitRetval = 0
CASE EventRecord["KEYCODE"] = -18 : ; if user pressed Alt-E
BEEP
WaitRetval = 1
ENDSWITCH

; now that all of possible events and triggers have been
; handled, change the prompt and the wait spec according
; to the current window
SWITCH
; if current window is the form, restore original prompt
; hide Main Paradox menu with blank menu
CASE GETWINDOW() = FormWindow :
PROMPT "Make changes to table, press [F2] when done"
SHOWPULLDOWN ENDMENU
NEWWAITSPEC TRIGGER "ARRIVEWINDOW"
KEY "F2", -18

; if user has opened the memo field, display appropriate
; prompt, replace the Editor menu with a SHOWPULLDOWN menu,
; and add the events MESSAGE "CLOSE" and Esc to wait spec
CASE ISFIELDVIEW() :
MemoWindow = GETWINDOW() ; get handle for memo field
PROMPT "Press [F2] to save changes to memo, Esc to cancel"
SHOWPULLDOWN
"DO-IT!" : "Save changes to memo field" : "Save",
"Cancel" : "Cancel changes to memo field" : ""
SUBMENU
"No" : "Continue editing the memo field" : "",
"Yes" : "Cancel changes, and close memo window" : "Cancel"
ENDSUBMENU
ENDMENU

```

```

NEWWAITSPEC TRIGGER "ARRIVEWINDOW"
MESSAGE "CLOSE", "MENSELECT"
KEY "F2", -18, "Esc"

ENDSWITCH

RETURN WaitRetval          ; now return the appropriate value to the WAIT

ENDPROC

COEDIT "Customer"
WINDOW HANDLE IMAGE 1 TO CustWindow      ; get handle for table view
WINDOW MOVE CustWindow TO 100,100       ; move table view off screen
PICKFORM 1
FormWindow = GETWINDOW()                 ; get handle for form window
WINDOW MOVE FormWindow TO 1,5           ; center form window

PROMPT "Make changes to table, press [F2] when done"
SHOWPULLDOWN                            ; hide Main Paradox menu
ENDMENU                                  ; with blank menu
WAIT WORKSPACE
PROC "WaitProc"
TRIGGER "ARRIVEWINDOW"                   ; original wait spec only
KEY "F2", -18                             ; includes ARRIVEWINDOW and
                                           ; the F2 and Alt-E key events

ENDWAIT

DO IT!
CLEARPULLDOWN                            ; remove SHOWPULLDOWN menu
CLEARALL

```



# Using multi-table forms

Multi-table forms help you quickly and easily develop applications that draw information from up to 10 tables simultaneously. Although the underlying structure of your database can be quite complex and the data spread across many separate tables, you can allow access to data through a simple multi-table form.

This chapter

- ❑ covers the rules for using multi-record and multi-table forms in applications
- ❑ presents the concepts of link keys and restricted views
- ❑ describes how to control user interactions with multi-table forms in the workspace
- ❑ discusses how to take advantage of Paradox's built-in support for maintaining referential integrity

---

## Basic concepts

To create a multi-table form, design forms for each of the tables you want to incorporate. Designate one of the tables a master table, depending on the relationship of the tables. Use Multi | Tables from the Form Designer menu to embed all other forms in the master form. You can embed up to 10 forms in the master. The embedded forms are called *detail forms*. The entire multi-table form belongs to the master table's family.

Master forms can have more than one page, and detail forms can be placed on any page of the master. Detail forms, however, can have only a single page. A detail form can have multiple records. The master form can have multiple records only if all the detail forms are unlinked.

A multi-table form can include both linked and unlinked detail forms.

- In the case of linked detail forms, there is a structured relationship between the data in the master and the details: The records in the master table control the records that are displayed in the linked detail tables. Changes made to the linked fields of the master table cause changes to the same (linked) fields in the detail table.
- In an unlinked detail form, there is no connection between the data shown in the master and the detail; changes made in one will have no effect on the other. Thus, the base table is a master only in the sense that it is the table with which the form is associated.

In your applications, you can use the `FORM()` function to determine the name of the current form, the `ISMULTIFORM()` function to determine whether a specified form is a multi-table form, and the `FORMTYPE()` function to discover if a form is a multi-record, linked, detail, or display-only form. The `FORMTABLES` command determines the names of the detail tables that are embedded in a master.

---

## Working with linked forms

Linked multi-table forms are very powerful tools you can use to develop applications that solve many common business problems with a minimum of coding. When used in conjunction with multi-record forms, linked multi-table forms let you maintain your data in separate tables while presenting it to users in a unified, easy to understand way.

Here are some of the features that make these forms so powerful:

- Built-in hot links let you display, edit, and enter related data in up to nine separate tables on a single form without having to do explicit joins or write complex update routines.
- Built-in referential integrity checking automatically maintains the integrity of data that is distributed among several tables—even in a multiuser context.
- Automatic scrolling regions let you display any number of detail records associated with a single master record.
- A large number of built-in functions help you manipulate multi-table forms and handle errors in your applications.

When a linked detail form is placed on a master form, the current record in the master table determines which records in the detail

table are displayed. Only those detail records whose link-key values match the values in the corresponding fields of the master are shown. Thus, whenever you view data through a linked detail form, you are seeing a restricted view of the data; at any time, you see only those records from the detail table that match or are associated with the current master record.

---

## Link keys

A *link key* is the part of the detail table's key that you match to fields of the master table as you design the multi-table form. All tables that you want to be linked detail tables must be keyed. In some cases, the link key might include all of the key fields of the detail; in other cases, it might comprise only some of them. The master itself must be a keyed table if there is a one-to-one or a one-to-many relationship (see the following explanation of possible relationships).

Paradox determines the exact relationship between the records in the master table and the detail tables through the relationship between link key fields and the corresponding fields of the master table. There are two rules about the placement of link key fields and their corresponding master fields on forms:

1. You cannot place the detail table's link key fields on its form.
2. You usually will want to place the corresponding linked master fields on the master form, but you aren't required to do so.

Master and detail records can be related to each other in one of four ways:

- **One-to-one:** Each master record has no more than one corresponding detail record. For example, in a personnel database, there can be a master record for each employee, and a corresponding health profile stored in another table. Each master employee record has no more than one corresponding record in the health profile table.

For a one-to-one relationship to exist, both tables must be identically keyed; the link key must be the detail table's entire key and the master table's entire key.

- **One-to-many:** Each master record has any number of corresponding detail records. For example, in an order entry system, each customer can place any number of orders. For each record in the master *Customer* table, there can be any number of corresponding orders in the *Orders* table.

In this case, the master table must be keyed. The link key of the detail table *must* contain all of the key fields of the master, but *cannot* contain all of the detail's key fields. The detail table's key must be comprised of more fields than the master table's key.

- Many-to-one: Any number of master records can correspond to a single detail record. For example, in a personnel system, many employees may work in a single department.

In this case, the master table may or may not be keyed or the detail may be linked to non-key fields. The link key of the detail table *cannot* contain all of the master table's key fields but *must* contain all of the detail's key fields.

- Many-to-many: Any number of master records correspond to any number of detail records. For instance, in a college student registration system, many students can each take many courses.

In a many-to-many relationship, the master table may or may not be keyed or the detail may be linked to non-key fields. The link key of the detail table *cannot* contain all of the detail or master table's key fields.

The following table summarizes these relationships:

Table 17-1 Link keys in multi-table forms

Relationship	Master keyed?	Link key contains entire master key?	Link key contains entire detail key?
One-to-one	Yes	Yes	Yes
One-to-many	Yes	Yes	No
Many-to-one	Yes or No	No	Yes
Many-to-many	Yes or No	No	No

You can use the PAL LINKTYPE() function to determine the type of link that the current detail table has to the master in a multi-table form (the FORMTYPE() function also provides information about a form). As the next example shows, you can use this information to control access to different parts of a multi-table form.

#### Example 17-1 Using LINKTYPE to distinguish parts of a form

Suppose you want to let the user specify a single master record to view; you want to use a multi-table form to present the master along with its associated detail records. To achieve this, you need to restrict the user from moving to another record while in the master image, while allowing the user to scroll to other records in the detail image. You can use the LINKTYPE() function to help achieve this effect.

This example assumes that the form has just one detail form placed on it, and you've already asked the user to specify a value to search for.

```
PROC WaitProc(TriggerType, EventRecord, CycleNumber)
```

```

    IF TriggerType = "DEPARTROW"           ; if user attempts to leave record
    THEN
        IF LINKTYPE() = "None"           ; if cursor is in master image
```

```

        THEN RETURN 1                ; break the current cycle -- don't
        ; allow user to move to another
        ; master record
        ELSE RETURN 0                ; otherwise, cursor must be in
        ; detail image - so process cycle
    ENDIF
ENDIF

IF TriggerType = "EVENT"            ; if F2, F3, or F4 was pressed
THEN
    IF EventRecord["KEYCODE"] = ASC("F2")
    THEN RETURN 2                    ; break wait if F2 pressed
    ELSE DOWNIMAGE                    ; otherwise F3 or F4 must have
    ; been pressed, so move to the
    ; other image-- this assumes only
    ; 1 detail is placed on the form
    RETURN 1                          ; don't process current cycle
    ENDIF
ENDIF
ENDPROC

PROC SingleMasterWait(TblName, FormName, SearchFld, SearchValue)

VIEW TblName                        ; put table on workspace
MOVE TO FIELD SearchFld              ; move to the field to LOCATE
LOCATE SearchValue                   ; look for requested value
IF NOT Retval                        ; value not found
THEN MESSAGE "No record matches the value you requested"
RETURN                               ; so quit
ENDIF
PICKFORM FormName                    ; go into multi-table form

PROMPT "[F3]/[F4] moves between Master and detail image, [F2] to quit"
SHOWPULLDOWN                         ; hide Main Paradox menu
ENDMENU                              ; with blank menu
WAIT WORKSPACE
PROC "WaitProc"                      ; call this procedure
TRIGGER "DEPARTROW"                  ; when user attempts to leave record
KEY "F2", "F3", "F4"                 ; or presses one of these keys
ENDWAIT

CLEARPULLDOWN                        ; remove SHOWPULLDOWN menu
CLEARIMAGE                            ; remove image when wait session ends

ENDPROC

; now call procedure, using form 3 of the
; Customer table, and search for 3266 in
; the Cust ID field
SingleMasterWait("Customer", 3, "Cust ID", 3266)

```

---

## Restricted views

A linked detail form provides a restricted view of the data in its table. That is, you see only those detail records whose link key values match the values in the associated fields of the current master record. A restricted view is like a miniature table—its image contains only the detail records for the current master. The image of the restricted view on the form follows all of the conventions for images on the Paradox workspace.

When you work with multi-table forms in your applications, you need to be aware of which view you are looking at from moment to

moment—at the entire table or just at a restricted view of it. You can easily work with both types of views in your applications if you remember the following simple rule:

- PAL commands and functions that operate on images respect the limits of restricted views; commands and functions that work on tables (that is, those that require you to explicitly name a specific table) do not.

Table 17-2 shows the commands and functions that respect the limits of restricted views when they are called in the context of a multi-table form (in all other contexts, they operate on the entire table).

Table 17-2 Restricted view commands and functions

<b>Keyword</b>	<b>Operation</b>
ATFIRST()	Tests if the current record is the first record of the restricted view
ATLAST()	Tests if the current record is the last record of the restricted view
BOT()	Tests for a move beyond the beginning of the restricted view after a MOVETO or SKIP command
DITTO	Copies the field from the previous record in the restricted view into the current blank field
END	Moves to the last record in the restricted view
EOT()	Tests for a move beyond the end of the restricted view after a MOVETO or SKIP command
HOME	Moves to the first record in the restricted view
IMAGEAVERAGE()	Average of the non-blank values in the current column of the restricted view
IMAGECOUNT()	Count of the non-blank values in the current column of the restricted view
IMAGECMAX()	Maximum value in the current column of the restricted view
IMAGECMIN()	Minimum value in the current column of the restricted view
IMAGECSUM()	Sum of the values in the current column of the restricted view
LOCATE	Locates a record in the restricted view
LOCATE INDEXORDER	Uses secondary index order to locate a record in the restricted view
MOVETO RECORD	Moves to a specified record in the restricted view
NIMAGERECORDS()	Returns the number of records in the restricted view
NROWS()	Returns the number of display rows in the restricted view

<b>Keyword</b>	<b>Operation</b>
RECNO()	Returns the record number with respect to the restricted view
ROWNO()	Returns the number of the current display row of the restricted view
SCAN	Steps through the restricted view, executing a sequence of commands at each record included in the view
SKIP	Moves the cursor forward or backward a specified number of records within the restricted view
WAIT TABLE	Lets the users examine or modify all the records in the restricted view
ZOOM	Moves to the first record with a specified value within the restricted view
ZOOMNEXT	Moves to the next record with a specified value within the restricted view

In addition to the HOME and END commands, all cursor movement commands (such as LEFT, RIGHT, PGDN, PGUP, DOWN, UP, and so forth) respect the limits of restricted views when used in that context

### Example 17-2 Using functions with restricted views

To see how a function works differently in different contexts, the following example calls IMAGECSUM() on both an entire table and on a restricted view of the same table. The code assumes that you have designed a multi-table form in which Customer is the master table and Orders is a linked detail.

```

VIEW "Orders"           ; place Orders table on the workspace
ECHO NORMAL            ; show user a snapshot of the workspace
ECHO OFF

MESSAGE LINKTYPE()    ; should return "None"
SLEEP 3000
MESSAGE ""             ; clear message
MOVETO [Quant]        ; go to the Quantity Ordered field
MESSAGE IMAGECSUM()   ; displays sum of values in the
SLEEP 3000             ; field for the entire table
MESSAGE ""

VIEW "Customer"       ; now put Customer on the workspace
PICKFORM 3            ; assumes that form 3 is multi-table form
                     ; with Orders as a linked detail
MOVETO "Orders"       ; move to the restricted view
                     ; for Orders on the multi-table form

MESSAGE LINKTYPE()    ; should return "1-M"
SLEEP 3000
MOVETO [Quant]
MESSAGE IMAGECSUM()   ; now IMAGECSUM() works on the restricted view--
SLEEP 3000            ; so now it displays the sum of values only for
MESSAGE ""            ; the current customer

```

You can easily use linked multi-table forms to present users with a selected subset of master records and their associated details. If a table is a master, you can query it to select a subset of records,

rename the *Answer* table to *Tempans*, and then use the COPYFORM command to copy its associated multi-table form to *Tempans*. Because of restricted views and the form's built-in links, when users page through the subset of master records shown in *Tempans*, they see only the corresponding subset of details, even though the entire detail table is referenced through the form.

---

## Multi-table forms and the Paradox workspace

When working with multi-table forms, there will be times when you need to toggle back to the Paradox workspace to do some background processing or computation or to reference a table not included on the form. It is therefore important to understand the relationship between the workspace and multi-table forms.

As the examples in this chapter have shown, to view a multi-table form, all you need to do is to place the master table with which the form is associated on the workspace and use either the PICKFORM or FORMKEY commands to go into form view. It is not necessary to explicitly place any of the detail tables included in the form on the workspace. However, there are times when you will want to place one or more of the detail tables or other tables on the workspace before displaying the form.

If you use the *WaitProc* procedure to move from a multi-table form to a table that is not embedded on the form, remember to use the FORMKEY command to toggle from form view into table view. You cannot issue a MOVETO to move directly from the form view of one table to the table view of another table; you must toggle into table view first. See Chapter 16 for more information about table navigation in the WAIT.

As you move between the workspace and a multi-table form, be aware that a few important commands used to navigate around the workspace operate slightly differently when they are used in the context of a multi-table form. Table 17-3 shows these differences.

Table 17-3 Commands to navigate the workspace

---

Command	Operation
UPIIMAGE/DOWNIMAGE	In table view or in a single table form, moves up or down to the next image on the workspace. In a multi-table form, cycles to the next image on the form.



Command	Operation
FORMKEY	In any context but a multi-table form, toggles between table and form view for the current table. In a multi-table form, toggles between the master table in table view and the master part of a multi-table form; from any form toggles back to table view of the master.
MOVETO	In table view, moves to any image on the workspace; in a multi-table form, moves to any image on the form.

To properly navigate around multi-table forms and between forms and the workspace, you should be aware of the following two rules:

- When you toggle from the master table to a multi-table form, the cursor will always be in the current record of the master form. When you toggle back to table view from a multi-table form, the cursor will be located in the current record of the master table no matter what image it was in during form view. This rule applies both to linked and unlinked detail forms.
- When you change master records or toggle between table and form view, the details associated with a master will always be displayed with the first record at the top of the group, and the first record is the current record. You can use the PAL command SETRECORDPOSITION to restore the position of a group of detail records within an image.

The following examples illustrate various interactions between multi-table forms and the workspace.

### Example 17-3 Functions on a multi-table form and the workspace

This example shows how you can summarize values in a field of both a restricted view and the entire table on the workspace. Each time the user moves to a new master record on a multi-table form, the wait procedure first moves to the image of the detail table included on the form and uses the function IMAGECSUM() to calculate the sum of items ordered by all customers. Because CSUM() does not require the table to be on the workspace, it's only necessary to view the master table; you don't have to explicitly put the detail on the workspace.

Showing the results of the calculation with a MESSAGE is a simple way to show column totals while the user is browsing around a table.

```
PROC WaitProc(EventType, EventRecord, CycleNumber)
    IF EventType = "ARRIVEROW"                ; if user moves to a new record,
    THEN                                       ; and the cursor is in the
        IF TABLE() = MasterTbl              ; master table...
        THEN
            MOVETO DetailTbl                  ; move to restricted view
            MOVETO FIELD DetailFld            ; field to calculate image sum
            Imsum = STRVAL(IMAGECSUM())       ; compute total for detail group
```

```

LOCK DetailTbl WL           ; write lock detail table for
                           ; CSUM() calculation
IF Retval
  THEN Totsum = STRVAL(CSUM(DetailTbl, DetailFld))
                           ; use CSUM() to compute total
                           ; for entire detail table
      UNLOCK DetailTbl WL   ; remove detail's write lock
  ELSE Totsum = "not available"
                           ; if write lock fails, don't
                           ; calculate CSUM(), tell user
                           ; total is not available
ENDIF
MOVETO MasterTbl           ; move back to master image

                           ; display customer statistics
MESSAGE "Customer has ordered ", Imsum, " items --",
        " Total items ordered is ", Totsum
ENDIF
RETURN 0                   ; return to wait -- do not
                           ; break cycle, or wait session
ELSE RETURN 2              ; event must be F2 key
ENDIF
ENDPROC

PROC ShowSums(MasterTbl, FormName, DetailTbl, DetailFld)
  VIEW MasterTbl           ; view master table
  PICKFORM FormName       ; assumes form is multi-table form with
                           ; one linked detail
  WINMAX                   ; maximize form window

  PROMPT "Press PgUp or PgDn to go to next record, F2 to quit"
  WAIT WORKSPACE          ; enter wait session
  PROC "WaitProc"
    TRIGGER "ARRIVEROW"   ; call wait procedure when user moves
    KEY "F2"              ; to another record, presses F2
  ENDWAIT

  CLEARIMAGE
ENDPROC

ShowSums("Customer", 3, "BookOrd", "Quant")

```

---

## Referential integrity

As you build applications that use data distributed among groups of related tables, you need to be aware of the issue of *referential integrity*. Referential integrity is a measure of the soundness of the links (or references) between tables that contain related data. Paradox contains a number of built-in checks that help to ensure referential integrity between tables that have one-to-one or one-to-many relationships and that are related through multi-table forms.

One of these checks prevents you from directly assigning values to the link keys of a detail table in a multi-table form. The values in the link key are controlled by the values in the associated fields in the master table. For tables that have one-to-one or one-to-many relationships, Paradox guarantees that if you change the linking values in a master record, the link key values in the corresponding

details are automatically updated. This prevents the links between the master and detail tables from being corrupted.

Paradox enforces this aspect of referential integrity in three ways:

- By definition, if a table is a linked detail embedded in a multi-table form, the fields comprising its link key cannot be placed on the form. Thus, in the context of the multi-table form, there is no way to assign values to those fields that might interfere with the link.
- If you begin to edit tables through a multi-table form in Edit or DataEntry mode, Paradox lets you make changes only through the form; if you toggle to table view, you will not be allowed to make any changes (in an application, you'll get error code 14 if you try). This is called *link-locking*. Link locks are not applied in Coedit mode.
- You can insert a new detail record with COPYFROMARRAY and append a new detail record with APPENDARRAY. The link values in the array must match the link key values in the target record; if not, a script error results.

In summary, the only way you can assign to or change the linked key fields of a linked detail table is by assigning or changing values in the linked fields of the master.

Even though Paradox automatically assigns values to the link keys of the detail, if a detail table is password protected, the user must have at least InsDel rights to the fields comprising the link key. If you don't want to give users update rights to the tables outside the context of an application, you can use the PASSWORD and UNPASSWORD commands to present and withdraw the appropriate access rights for the duration of the update routine.

In addition to making sure that the link keys of related tables are protected, Paradox automatically prevents interactive users and PAL applications from taking certain other actions that could undermine referential integrity. For example, if you are editing data through a multi-table form, you cannot delete a master record while linked details depend upon it, even with POSTRECORD FORCEPOST. You must first explicitly delete the detail records before Paradox will let you delete the master (for similar reasons, DOS will not let you delete a directory if it still contains files or subdirectories). If you try to delete a master without deleting the linked detail records, you will cause an error (errorcode 14).

---

## The blank key problem

Another issue connected with referential integrity concerns what might be called the "blank key problem." Because Paradox uses link key values to associate detail records with their master, if a blank

exists on either side, it could cause problems. For example, suppose a master table and its corresponding detail table both have unrelated records with blank values in their link key fields. As soon as the tables are defined to have a relationship through a multi-table form, the blank master and details immediately become associated with each other, whether they really belong together or not. If you then change the key value in the master, Paradox automatically updates the link key of the details, thus making it even harder to dissociate the details from the master.

Because of such problems, Paradox requires you to fill in the key of a new master record before you enter the details. In your applications, when users are entering new data using multi-table forms, you should make sure that the first action for each new master record is to assign a unique value to the key fields. Then when the user begins to fill in the detail table, there will be a unique key for the corresponding detail records.

---

## **Ensuring referential integrity**

Because referential integrity in Paradox is enforced through multi-table forms, it is important that you make sure that your users can access related tables only through the appropriate form when those tables are involved in update and data entry operations.

You can do this by applying the appropriate password protection to each of the tables involved in an application. In general, you'll want to reserve the passwords that grant update and entry rights for use internally in your scripts. Also, you should make sure to restrict users' rights to modify the multi-table forms associated with the master tables used in your application.

All of Paradox's referential integrity checking works in a multiuser as well as a single-user context. Chapter 23, "Multiuser applications," describes how to build multiuser applications in general and also contains a discussion of some of the special issues related to using multi-table forms on a network.

# Controlling Paradox

Good programming involves picking the best method available to implement the task you're automating. That's what this section of the manual will help you do. The four chapters in this part cover the following topics:

- ❑ Chapter 18, "Interacting with Paradox." This chapter reviews the PAL commands available to control operations on the Paradox workspace.
- ❑ Chapter 19, "Manipulating values." Chapter 3 introduced field specifiers, variables, and arrays; this chapter explains how to use them to manipulate values, placing special focus on COPYTOARRAY, COPYFROMARRAY, and query variables.
- ❑ Chapter 20, "Keyboard macros." One handy productivity tool, both in creating applications and in using them, is the creation of single-key macros to perform larger tasks. This chapter shows you how to create and use them.
- ❑ Chapter 21, "Performance and resource tuning." This chapter explains how Paradox manages memory within a PAL application, and how you can fine-tune your application to run as quickly as possible.



# Interacting with Paradox

We've described PAL as an "automated Paradox user" that can operate Paradox for you and for users of your scripts and applications. This chapter provides a detailed explanation of how to get PAL to manipulate Paradox in your scripts. It covers

- how to control Paradox modes
- recorded keypress instructions
- simulated keypress instructions
- when to use the various Paradox options for locating data
- how to apply statistical and business analysis functions to columns of data

Much of the material in this chapter summarizes and extends the basic building blocks covered in Part I of this manual. Chapter 2, for example, introduced recorded scripts and explained how keypresses and menu choices are recorded in them. Here we'll expand on those building blocks by introducing ways to simulate and compute keypresses and menu choices. We'll also focus on the big picture and show how you can make a single application or script from a combination of recorded keypresses, simulated keypresses, and PAL programming commands.

---

## How modes determine what you can do

At any point, Paradox is in one of the following major modes:

- |                                    |                                      |
|------------------------------------|--------------------------------------|
| <input type="checkbox"/> Coedit    | <input type="checkbox"/> Main        |
| <input type="checkbox"/> Create    | <input type="checkbox"/> Password    |
| <input type="checkbox"/> DataEntry | <input type="checkbox"/> Report      |
| <input type="checkbox"/> Edit      | <input type="checkbox"/> Restructure |

- Form
- Graph
- File editor
- Index
- Script
- Sort
- Preview
- SetConn (for SQLLink)

The structure of Paradox is basically flat: a two-level hierarchy with Main mode at the top on the first level, and all the remaining modes subordinate to it on the second level (although Preview mode is also available from Report mode). When you use Paradox interactively, you always start out in Main mode, and you must be in Main mode to access any of the other modes except File editor and Script.

You change modes by making menu choices or issuing commands that have the side effect of moving from one mode to another. In some contexts, a particular command can have different consequences in terms of moving between modes. For instance, all queries are executed in Main mode. When you invoke the Do\_It! command, it performs the current query statement, and Paradox remains in Main mode.

If you invoke Do\_It! in any other mode, it completes the current operation and returns Paradox to the mode you were in previously.

Different windows can have different modes. For example, when you view a table from the Main menu, the table window is in Main mode. If you then select Scripts|Editor|Open and begin to edit a script, the script window is in Script mode. If you select File|Open and begin to edit a text file, you have a third window open that is in File Editor mode. Selecting each of these different windows then has the effect of also changing the mode that Paradox is in.

Some commands have the effect of changing modes once. For example, EDIT changes Paradox from Main mode to Edit mode. Other commands have the effect of changing modes twice. For instance, SORT changes Paradox from Main to Sort mode, performs a sort, and then changes Paradox back into Main mode.

Which PAL commands are valid at any given moment depends upon the Paradox mode. For example, in Main mode, you can't use a command like

```
[Stock #] = "45-ABC-78"
```

to make a field assignment to the current table. To make field assignments, you must be in Edit, DataEntry, or Coedit mode. Trying to operate on a table in the wrong mode is one of the most common errors made by PAL programmers.



---

## The current mode

When using Paradox interactively, the current mode is highlighted in the lower right corner of the screen. Also, each mode has its own menu. If your script needs to know what mode it's in, you can use the `SYSMODE()` function to find out.

Once you know which mode you're in, how do you know what you can and can't do? For example, when you say

```
VIEW "Orders"
```

you are asking Paradox to put the *Orders* table on the workspace. But when can you put a new table on the workspace? Obviously, you can't do it while you're in the Script Editor (Script mode). Less obviously, you can't do it while you're in Edit or Coedit mode.

The simplest answer is really another question: What does interactive Paradox allow you to do at the moment? If you're viewing a table and want to change it, how would you do that interactively? You'd either press Menu *F10* and choose Modify | Edit, or you'd simply press Edit *F9*. In a script, you'd do the same thing:

```
Menu {Modify} {Edit} {Orders}
```

or the equivalent abbreviated menu command

```
EDIT "Orders"
```

or if *Orders* is already on the desktop, simply

```
EDITKEY
```

Similarly, if you were editing the *Orders* table and wanted to place another table on the workspace, how would you do it interactively? First, you'd have to get out of Edit mode by pressing Do-It! *F2* or choosing DO-IT!, or Cancel from the Edit menu. You would then be back in Main mode where you can press Menu *F10* to display the Main menu and then choose View.

As we've emphasized before, it is important to know Paradox to use PAL effectively. If you know how Paradox reacts at any point, you'll know what you can and can't do at that point with PAL commands. Many of the descriptions of individual commands in the *PAL Reference* tell you the mode or modes that Paradox must be in for the command to work.

---

## Minor modes

In addition to the major modes described previously, there are minor modes, or submodes, that govern what can and cannot be done at a particular time. There are built-in PAL functions that let you test whether Paradox is in one of these minor modes.

The four minor modes and their corresponding functions are

- Insert or overwrite (`ISINSERTMODE()`)

- ❑ Field view (ISFIELDVIEW())
- ❑ Help (HELPMODE())
- ❑ Form view (ISFORMVIEW())

Changing from one minor mode to another is a side effect of a command or keypress. For example, pressing Form Toggle *F7* or invoking the FORMKEY command has the effect of going into form view for a table on the workspace.

It's important to understand the way Paradox operates in the various minor modes. For example, in form view, certain cursor movement commands work differently than they do in table view—*PgUp* moves the cursor up to the previous record in form view, but up a screenful of records in table view. As with the major modes, the best way to gain this understanding is to work with and become familiar with the way Paradox works interactively.

## Using recorded keypress interaction

The simplest way to control Paradox from a script is to record sequences of keypresses interactively. When you use Scripts | Begin Record or Instant Script Record *F3* to record a script, the keys you press are recorded in one of three ways:

- ❑ Menu commands appear in braces, such as {Create}.
- ❑ Special Paradox keys appear as key names, such as Tab, Right, or Do\_It!.
- ❑ Entries you type in a field or in response to a prompt appear as quoted strings, such as "Hotel Cairo" or "Stock #".

Knowing this, there is nothing that says you can't use the same command sequences in your scripts. Instead of recording a script that creates a new table with the fields Stock #, Description, and Price, you could use the Editor to type the same sequence Paradox would record:

```
{Create} {procline} "Stock #" Tab "N" Enter
"Description" Tab "A25" Enter
"Price" Tab "$" Do_It!
```

All three types of recorded keypresses are used in this script: {Create} is a braced menu command, "Stock #" is a quoted string, and Do\_It! is a special Paradox key. This script assumes that the Paradox Main menu is already displayed when it begins execution.

To create a script from scratch that mimics a Paradox recorded script requires you to know—and reproduce from memory—just how Paradox operates. That's why it's usually simpler to record

keystrokes. For instance, to recreate this example from scratch, you would have to know exactly how to create a new table:

1. Choose Create from the Paradox Main menu.
2. Type the name of the table to create.
3. For each field in the table structure, type the field name, press *Tab*, type the field type, and press *Enter*.
4. Press *Do\_It!* *F2* to record the structure and complete the table creation.

There is a hidden problem with this example, though; step 2 assumes that the table name you type doesn't already exist. If it exists, Paradox prompts you that the name has already been used.

In this case, our script (whether recorded or typed in) wouldn't work.

This is one reason why the PAL abbreviated menu command CREATE does not ask for verification to replace existing tables. This is also why it pays to search carefully for potential trouble areas in recorded scripts. There are at least two ways to safeguard this table creation routine to make sure that existing tables don't invalidate it:

- One way is to check to see if the table exists before allowing the create instruction:

```
IF ISTABLE("Prodline")
  THEN MESSAGE "Table exists, cannot create it"
  ELSE {Create} {Prodline}
    "Stock #" Tab "N" Enter
    "Description" Tab "A25" Enter
    "Price" Tab "$" Do_It!
ENDIF
```

You might want to make the table name a variable so that you can get a new name from the user if the ISTABLE() function returns True. (We've done this in the next example.)

- A better way of solving the existing table problem is to use more of the capabilities of the PAL command set to recast it. For example,

```
TblName = "ProdLine"
WHILE ISTABLE(TblName)
  BEEP ; let user know a response is needed
      ; create a window for user input
  WINDOW CREATE FLOATING WIDTH 50 HEIGHT 5 TO InputWindow
  @ 1,1 ?? TblName, " exists. Type new table name: "
  ACCEPT "A8" REQUIRED TO TblName ; get user's response
  WINDOW CLOSE
ENDWHILE
CREATE TblName "Stock #" : "N", "Description" : "A25",
  "Price" : "$"
```

This example uses three techniques to avoid the existing table prompt:

- ❑ It converts the table name into a variable.
- ❑ It uses a WHILE loop to make sure the table name doesn't already exist.
- ❑ It replaces the recorded {Create} sequence with the abbreviated menu command CREATE.

Don't get the idea that there is anything wrong with reproducing or retaining sequences of recorded keystrokes in your final applications. In fact, there are certain operations, such as importing data to and exporting data from Paradox, that can't be accomplished any other way.

However, in order to build robust applications, you need to make sure that you step through every possibility at each point in a recorded sequence. If there is a potential for ambiguity, such as when you try to create a table that already exists, you should enhance the flow of control to make sure that the script does exactly what you want it to do.

For instance, if you do quite a bit of table creation, copying, or renaming in your application, you would probably want to make the whole WHILE construct from the last example into a procedure:

```
PROC GetTableName(Tb1Name)                ; Tb1Name is private
  WHILE !STABLE(Tb1Name)
    BEEP                                   ; let user know a response is needed
                                           ; create a window for user input
    WINDOW CREATE FLOATING WIDTH 50 HEIGHT 5 TO InputWindow
    @ 1,1 ?? Tb1Name, " exists. Type new table name: "
    ACCEPT "A8" REQUIRED TO Tb1Name        ; get user's response
    WINDOW CLOSE
  ENDWHILE
  RETURN Tb1Name
ENDPROC
```

The use of a procedure to check the table name is more versatile (it can be used with operations other than CREATE) and reduces the code necessary to perform a table name check to the single line:

```
(GetTableName(tb1name))
```

Now you can call the *GetTableName()* procedure before any command or sequence that might replace an existing table:

```
tb1 = GetTableName("Prodline")
CREATE tb1
  "Stock #":"N",
  "Description":"A25",
  "Price":"$"
```

Query forms are a breed apart; in almost no circumstance should you use the Script Editor to reproduce the recorded form of a query image. Composing the query interactively in Paradox and using Scripts | QuerySave to store the image in a script ensures that the script contains a valid query statement.

---

## Simulating keypress interaction

---

### Abbreviated menu commands

The PAL abbreviated menu commands offer a powerful way to reproduce the effect of sequences of recorded keystrokes. These commands don't do anything you can't do with recorded keystrokes; in fact, they exactly reproduce the actions that would take place in Paradox if you made the equivalent menu choices interactively. You can test this by using `Menu {Scripts}{ShowPlay}` to play a script containing abbreviated menu choices.

Abbreviated menu commands are simpler to type into a script than the equivalent sequence of keystrokes. For example,

```
RENAME "Oldtab" "Newtab"
```

is the same as the recorded script

```
Menu {Tools} {Rename} {Table} {Oldtab} {Newtab}
```

We suggest that you use abbreviated menu commands wherever possible, since they have three advantages over sequences of keypresses:

- ❑ Abbreviated menu commands do not require the user to confirm that an action is to be taken as do many of the equivalent sequences of interactive menu choices.
- ❑ You can easily set parameters for operations by using variables as arguments to abbreviated menu commands. While you can get the same effect by modifying keypress sequences (as shown in the next section), it is not as easy.
- ❑ Abbreviated menu commands are easier to read and follow.

---

### SELECT, KEYPRESS, and TYPEIN

As noted previously, there are certain keystroke sequences (such as importing and exporting data) for which no abbreviated menu commands exist. In these instances, it's essential to be able to set parameters for an operation, even though you can't normally include a variable in place of a braced menu command, special Paradox key, or quoted string. For example, the following sequence, which is designed to import an arbitrary Quattro Pro file into a Paradox table, results in a script error:

```
pdxtable = "Orders"  
QPfile = "QPOrders"  
{Tools} {ExportImport} {Import} {Quattro}  
{2} Quattro Pro} {QPfile} {pdxtable} ; this won't work!
```

Instead of substituting the values "QPOrders" and "Orders" for the variables *QPfile* and *pdxtable*, Paradox looks for a Quattro Pro file called *QPfile* and a Paradox table called *pdxtable*.

To handle such cases, three PAL commands simulate the effect of interactive keypresses:

- SELECT imitates choosing menu commands (you don't need to use *Enter* after a SELECT).
- KEYPRESS imitates a single Paradox keystroke.
- TYPEIN imitates multiple keystrokes in Paradox.

Each of these commands takes an expression as an argument, and that expression can include variables. For example, the Quattro Pro import script will work if it is rewritten as follows:

```
pdxtable = "Orders"
QPFile = "QPOrders"
{Tools} {ExportImport} {Import} {Quattro}
(2) Quattro Pro)
SELECT QPFile ; this works!
SELECT pdxtable
```

Since these keypress commands let you substitute variables wherever you want, you could use this ability, for example, to create a table, given a variable (*tblname*) and two arrays (*flds* and *types*) which contain its name, field names, and field types, respectively:

```
(Create) ; request to create a table
SELECT tblname
FOR Counter FROM 1 TO ARRAYSIZE(flds) ; now enter fields for the table
; flds array contains field names
; types array contains matching field types
TYPEIN flds[Counter] Tab TYPEIN types[Counter] Enter
ENDFOR
Do_It! ; process the request
```

You could not reproduce this effect with either straight keypress interaction or the CREATE command.

As with recorded keypresses, there are some restrictions on when you can use the keypress commands:

- SELECT works only when a menu or menu prompt would be active on the workspace.
- TYPEIN works only when Paradox is either waiting for the user to enter a response or a value.
- KEYPRESS works only when Paradox can accept the keystroke you specify.

If you use one of these commands in the wrong context, a script error results.

If these commands don't suit your needs, try the EXECUTE command (see EXECUTE in the *PAL Reference*).

---

## Working with tables

One of the advantages of working with Paradox is that it lets you interact at a high level with tabular data structures. It's easy to understand how to manipulate these structures with PAL because they are nothing more than regular Paradox tables. You work with them in PAL much as you would use them interactively in Paradox. In order to use the data stored in a table, all you need to do is to place that table on the Paradox workspace. To write a clean application, you must always keep track of what tables are on the workspace, what order they're in, which one is current, and so on.

---

### Locating records

To work with specific values in a table, you generally need to move to the record that contains them. If you know the record number, you can use the MOVETO command to move directly to that record. But more often you need to find a particular value or set of values without knowing exactly where they are located. In addition, if other users are coediting the table, the record numbers are constantly changing. There are four ways to find records by value:

- The LOCATE command finds a match in the current field or in the leftmost two or more fields. It is useful when you want to move through a table in the base table order, stopping at each matching record, perhaps to perform some operation on the record for that field.
- The LOCATE INDEXORDER command finds a match by using the secondary index order of the table in the current field. It is useful when you want to move through a table in the secondary index order, stopping at each matching record, perhaps to perform some operation on the record for that field.
- The ZOOM command, like *Zoom Ctrl-Z*, is a shortcut for Menu {Image} {Zoom} {Value}. It moves the cursor to the first record that matches a value in the current field. You can also use ZOOMNEXT, like *Zoom Next Alt-Z*, to move to the next matching record.
- A FIND query moves the cursor to the first record that meets your selection criteria. In addition, it puts all records that meet the criteria in the *Answer* table. FIND puts the queried table on the workspace if it's not already there; it does not put *Answer* on the workspace, however.

These four methods of locating records are summarized in Table 18-1.

Table 18-1 Comparison of record location methods

Feature	LOCATE	LOCATE INDEXORDER	ZOOM	FIND query
Search for exact matches	Yes	Yes	Yes	Yes
Search for wildcard patterns	Yes	Yes	Yes	Yes
Fields searched	Single (current) field or multiple (first <i>n</i> ) fields	Single or multiple fields defined by secondary index	Single (current)	Multiple (any fields)
Search for next/prev match	NEXT keyword only	NEXT and PREV keywords	ZOOMNEXT	In <i>Answer</i> table
Search for first/last match	No	FIRST and LAST keywords	No	No
Availability	PAL command	PAL command	PAL command Menu command Keypress	Query
Modes of operation	Main, Edit, Coedit	Main, Edit, Coedit	Main, Edit, Coedit	Main
Uses indexes	Main and Coedit only	Main and Coedit only	Yes	Yes
Sets <i>Retval</i> if value located	Yes	Yes	No	No

## Looking up and storing values

Once you've found the record you want, there are two main PAL techniques for getting information into or out of it:

- using variables with field specifiers
- using arrays with COPYTOARRAY and COPYFROMARRAY

### Field specifiers

Field specifiers, introduced in Chapter 3, are handy for working with a single field of a record in a table. You can use field specifiers on either side of an assignment statement:

```
[Stock #] = "45-ABC-33" ; must be able to edit table
PtNum = [Stock #] ; can be in Main mode, with table on workspace
```

The first example replaces the current value in the *Stock #* field with the value "45-ABC-33". The second example gets the value in the *Stock #* field and stores it in the variable *PtNum*.

Other examples of field specifiers include

```
[ ] = "45-ABC-33" ; assign value to current field
CurRec = [#] ; assign a current record number to CurRec
PtNum = [Orders -> Stock #] ; assign PtNum the value in the Stock #
; field of current record of Orders table
[Orders->] = "45-ABC-33" ; assign value to current field of Orders
[Orders(Q)->] = "45-ABC-33" ; assign value to current field
; of Orders query image
```



---

## Arrays and record values

You can use three special commands to work with entire records of information at a time:

- COPYTOARRAY creates an array that contains the entire current record of the current table.
- COPYFROMARRAY creates (or replaces) the current record in the current table using the data stored in an array.
- APPENDARRAY appends a record to the end of the current table using the data stored in an array.

For example, here's how you might use COPYTOARRAY and COPYFROMARRAY to copy the first record of one table into the last record of another, identically structured table:

```
tab1 = "Orders" ; table to receive new record
tab2 = "Newords" ; table that contains record to transfer
VIEW Tab2 ; retrieve second table
COPYTOARRAY a ; copy first record into array a
EDIT tab1 ; now get first table ready for changes
END ; get to last record
DOWN ; add a new, blank record
COPYFROMARRAY a ; and copy info from second table to new record
```

You can also use field specifiers with arrays that are dynamically created with COPYTOARRAY. For instance, if you really wanted only to grab the Stock # from *Newords* and put it into the 10th record, you could use

```
Tab1 = "Orders"
Tab2 = "Newords"
VIEW Tab2 ; retrieve second table
COPYTOARRAY a ; copy first record into array a
EDIT Tab1 ; now get first table ready for changes
MOVETO RECORD 10 ; move to 10th record
[Stock #] = a["Stock #"] ; assigns Stock # from Stock # field of array
```

COPYTOARRAY, COPYFROMARRAY, and APPENDARRAY are described in detail in Chapter 19 and in the *PAL Reference*.

---

## Checking the desktop status

Your script can report information to you about the status of the desktop. Table 18-2 lists the status functions and commands that let you find out what is happening on the desktop. Example 18-1 shows how you might use some of them in an application.

As you can see, the status commands and functions can help you sort out what is happening on the desktop. They are especially helpful in scripts after a user has used Paradox interactively, or when control is passed to your script from another script which leaves the workspace in an indeterminate state. Remember that you can always use the RESET command to restore Paradox to a known state, CLEARALL to

clear all the image windows from the workspace, or ALTSPACE {Desktop} {Empty} to close all windows except floating canvases.

The operation of some of the status functions is slightly altered when they are called in the context of a multi-table form. See Chapter 17 for details.

Table 18-2 PAL status commands and functions

Function	Description
ATFIRST()	Is current record the first record in table?
ATLAST()	Is current record the last record in table?
BANDINFO()	What is current band in a report spec?
BOT()	Tests for move beyond the beginning of a table
COL()	What screen column is the cursor in?
COLNO()	What is current column number of an image?
CURSORCHAR()	What is current character at the cursor in the Designers?
CURSORLINE()	What is current line at the cursor in the Designers?
EDITOR INFO	Creates a dynamic array with information about the current Editor session
EOT()	Tests for move beyond the end of a table
ERRORINFO	Creates a dynamic array with information about the latest script error
FIELD()	What is the name of the current field?
FIELDINFO()	What is the current field (in Form or Report mode)?
FIELDNO()	What is the number of the field?
FIELDTYPE()	What is the data type of the current field?
FORM()	What is the active form for the current table?
FORMTYPE()	What kind of form is the current form (MultiRecord, Linked, Detail, or DisplayOnly)?
GETCANVAS()	Returns the handle for the window that displays current canvas painting commands
GETCOLORS	Creates a dynamic array containing attributes of the current color palette
GETKEYBOARDSTATE	Reads the current keyboard attributes into a dynamic array
GETWINDOW()	Returns the handle for the current window
GRAPHTYPE()	What is the current graph type?
HELPMODE()	Is user viewing Help or Lookup Help?
IMAGENO()	What is the number of the current image?
IMAGETYPE()	What type is the current image?
ISEMPTY()	Is a table empty?

<b>Function</b>	<b>Description</b>
ISFIELDVIEW()	Is current field in field view?
ISFORMVIEW()	Is current image in form view?
ISINSERTMODE()	Is Paradox in insert mode?
ISLINKLOCKED()	Is the current table protected from editing outside a multi-table form?
ISMULTIFORM()	Does a form have embedded tables?
ISMULTIREPORT()	Does a report have linked lookup tables?
ISWINDOW()	Tests whether the argument evaluates to the handle of an active window
LINKTYPE()	What is the relationship between the current embedded table and its master?
MENUCHOICE()	What is the current highlighted menu choice?
MENUPROMPT()	Returns the text of the current type-in prompt
NFIELDS()	How many fields are in a table?
NKEYFIELDS()	How many key fields are in a table?
NIMAGERECORDS()	How many records are in the current image on the workspace?
NIMAGES()	How many images are on the workspace?
NPAGES()	How many pages are in the current form or report?
NRECORDS()	How many records are in a table?
NROWS()	How many rows are visible in the current image?
PAGENO()	What is the current page in form or report?
PAGEWIDTH()	What is the page width in the current report?
QUERYORDER()	How are fields ordered in <i>Answer</i> ?
RECNO()	What is the current record number in the table?
RECORDSTATUS()	What is the condition of the current record (New, Locked, Modified, or KeyViol)?
ROW()	What screen row is the cursor in?
ROWNO()	What is the current row number in an image or multi-table form?
SYSINFO	Creates a dynamic array with information about the system running Paradox
SYSMODE()	What mode is Paradox currently in?
TABLE()	What is the name of the current table?
WINDOW()	What text is displayed in the Paradox message window?
WINDOW GETATTRIBUTES	Creates a dynamic array with elements that represent the attributes of a window
WINDOW GETCOLORS	Creates a dynamic array containing the color palette attributes for the specified window
WINDOW HANDLE	Gets a handle for the specified window

Function	Description
WINDOW LIST	Creates a fixed array with the window handles of all windows presently on the desktop
WINDOWAT()	Returns the handle of the topmost window that contains specified screen coordinates

### Example 18-1 Using status functions

Suppose you need to know if an image is already present on the workspace. You can use workspace status functions to find out.

```

WHILE (IMAGENO() > 1) ; get to first image
  UpImage
ENDWHILE

FOR i FROM IMAGENO() TO NIMAGES()
  IF tblname = TABLE()
    THEN ? "Table is already on workspace"
  ENDF
ENDFOR

```

## Column calculations

As shown in Table 18-3, PAL has a wealth of handy calculation functions that operate on an entire column of data at once. These functions are a fast method of doing calculations on a field, and save you from having to write a controlling loop to grab and increment a field value for each record. Remember, queries can also perform speedy calculations.

For example, suppose you want to guarantee that each entry in an Order # column represents the next sequential order number. It's easy to do this using the CMAX() function. All you need to do is add a line like this to your data entry module:

```
[Order #] = CMAX(TABLE(), [Order #]) + 1
```

Table 18-3 Column calculation functions

Function	Description
CAVERAGE()	Average of all non-blank values in a column
CCOUNT()	Number of non-blank values in a column
CMAX()	Largest value in a column
CMIN()	Smallest value in a column
CNPV()	Net present value of the non-blank series of cash flows in a column
CSTD()	Standard deviation of the non-blank values in a column

<b>Function</b>	<b>Description</b>
CSUM()	Sum of all non-blank values in a column
CVAR()	Variance of all non-blank values in a column
IMAGECAVERAGE()	Average of all non-blank values in a column of the current image or restricted view
IMAGECCOUNT()	Number of non-blank values in a column of the current image or restricted view
IMAGECMAX()	Largest value in a column of the current image or restricted view
IMAGECMIN()	Smallest value in a column of the current image or restricted view
IMAGECSUM()	Sum of all values in a column of the current image or restricted view

These column calculation functions make it easy to create a statistical report on a numeric field with PAL, since you can let the functions do the work and worry only about the format. All the functions in Table 18-3 place a write lock on the table until they have completed.

Functions with names beginning with IMAGE work on the current column of the current image on the workspace. As described in Chapter 17, these functions are useful when you want to do calculations based upon a restricted view of the data in a multi-table form.

The other column calculation functions require you to explicitly specify the table and column in which you want to do the calculation. It is not necessary for the table named in the call to be on the workspace when the function is invoked.

## Example 18-2 Calculating test statistics

You could use this short PAL script to summarize the information in a field called Test Scores:

```
FieldName = "Test Scores"
TableName = "TestData"
MaxScore = CMAX(TableName, FieldName)      ; store calculation result
MinScore = CMIN(TableName, FieldName)      ; in a variable
AvgScore = CAVERAGE(TableName, FieldName)
VarScore = CVAR(TableName, FieldName)
StdScore = CSTD(TableName, FieldName)
CntScore = CCOUNT(TableName, FieldName)

                                ; create a window for output
WINDOW CREATE @4,14 WIDTH 50 HEIGHT 12 TO DisplayWin
SETCANVAS DisplayWin           ; make DisplayWin current canvas
@1,0 ?? FORMAT("W48,AC", "Test Score Summary")
@2,0 ?? FORMAT("W48,AC", STRVAL(CntScore) + " students took the test")
SETMARGIN 8                    ; text begins at column 9

                                ; display each statistic in the
                                ; DisplayWin window
? "The lowest score was: ", FORMAT("W8.1,AR", MinScore)
? "The highest score was: ", FORMAT("W8.1,AR", MaxScore)
? "The average score was: ", FORMAT("W8.1,AR", AvgScore)
? "The std. deviation was: ", FORMAT("W8.1,AR", StdScore)
? "The score variance was: ", FORMAT("W8.1,AR", VarScore)
```

In this example, you did nothing but tell Paradox what you wanted to have calculated and how you wanted it displayed. It was not even necessary to place the table on the workspace. In addition to saving you from writing your own complex calculation routines, PAL's column calculation functions are swift, requiring only a single pass through the database for each.

If you need to get summary information about a portion of a file—that is, not all the records in the table—the best approach is to create a query that isolates the records you want to summarize. Include a CALC in the query that uses the statistical functions so Paradox selects records and computes the statistical function at the same time.

# Manipulating values

This chapter discusses how you can use PAL to manipulate values. When you use Paradox interactively, you mainly work with information in tables. You can add to, change, or delete that information, but there is no easy way to hold or manipulate it temporarily.

PAL excels in manipulating temporary information. With field specifiers, variables, and arrays, you can store, manipulate, and replace data in your tables without having to work directly on the table. You can add validity checks, convert or summarize raw values before storing them, or even request that the user type something more than once to ensure that the values you capture are correct. This chapter explains how to use five of PAL's extensions to Paradox that let you manipulate values:

- field specifiers
- variables
- fixed arrays
- dynamic arrays
- query variables

---

## Field specifiers

As described in Chapters 3 and 18, you can use PAL field specifiers to reference fields in table and query images. You can both read and write table values with field specifiers. For example, if the current image is the *Orders* table and the current field is *Stock #*, you can use any one of the following statements to assign its value in the current record to the variable *partnum*:

```
partnum = [] ; current field in current image
partnum = [Stock #] ; named field in current image
partnum = [Orders -> Stock #] ; both image and field specified
```

Conversely, you can use field specifiers to assign values *to* fields. You must be in Edit or Coedit mode to do this:

```
[] = partnum ; value of partnum is assigned to current field
```

But you don't need to read table values into variables to manipulate them. In essence, you can use field specifiers to work with table values in place. In the *Products* table, for example, this command increases the Price field in every record by 10%:

```
COEDIT "Products"
SCAN
  [Price] = [Price] * 1.1
ENDSCAN
```

---

## Using variables

As in most programming languages, you can use variables in PAL to hold and manipulate information temporarily. A variable is like a mailbox that holds one item of information.

If you are an experienced programmer, you may find PAL's use of variables comparable to the way variables are treated in other programming languages.

Following are the features of variables that are likely to be similar to the way variables are treated in other languages:

- You can name variables yourself.
- You assign variables by putting the variable name to the left of an equal sign and an expression to the right, like this:

```
CurMonth = MOY(TODAY()) ; assigns current month to CurMonth
```

The value of the expression `MOY(TODAY())` in the above example is assigned to the variable *CurMonth*.

The following list is what may be different about the way PAL handles variables:

- You don't have to explicitly declare variables prior to use.
- Variables are not assigned data types for the duration of a program, but rather are assigned data types at any given time according to the data they hold.
- Variables are globally available to the entire PAL environment unless:
  - They are formal parameters to a procedure.



- They are defined as PRIVATE to a procedure.
- They are defined within the scope of a closed procedure.

Even variables private to a procedure are global to other procedures called by it. This scheme is called *dynamic scoping* of variables. Global variables must explicitly be passed to a closed procedure with the USEVARS keyword.

- Once a global variable has been assigned a value, it retains that value throughout the Paradox session until it is explicitly reassigned or released.

The implications of these differences are discussed in the next sections.

---

## Lifetime of variables

New PAL programmers often forget that variables can still be defined from a previous script when they play a new one. Thus, to make sure that your applications start with a clean slate, either

- Put a RELEASE VARS ALL statement at the beginning of the top level application script.
- Or structure your applications using closed procedures so that variables are automatically released when the application terminates.

You should also be concerned about variables remaining active across scripts if you are writing modular applications. There are several ways to resolve this issue:

- The best way is to make the modules into closed procedures. This way, there is no need to be concerned about the variables in one module spilling over to the other modules.

When you need to pass variables between closed procedures, you can use the USEVARS keyword.

- For regular procedures, you can restrict the scope of variables by using the PRIVATE keyword. PRIVATE variables are automatically released when the procedure ends.
- If you're not using a closed procedure at the top level of your application, unless you need to pass global variable values back to Paradox, put a RELEASE VARS ALL command at the end of your application.
- Choose variable names that are unlikely to be duplicated. For example, in a module called LOOKUP that looks up a value in a table, you might use names that start with LU\_ for those variables that are available globally, such as *LU\_Changed*, *LU\_LastChar*, and so on. Since users or other applications are unlikely to start their

own variable names the same way, this assignment of variables will probably not interfere with another script variable.

One word of caution about dynamic scoping: When you declare a variable private to a procedure, it is only available within that procedure. But any other procedures called by the first procedure have access to the private variables of the first procedure. In other words, a variable declared private is available to the procedure that declared it, and globally to any other procedures called within that procedure. See “Variables and procedures” in Chapter 6 for more information.

---

## Typing variables

Ad hoc typing of variables is also important to understand. Most programming languages “type” variables to contain a certain kind of information (numeric, date, alphanumeric, or so on). Once a variable is typed for one kind of data, that is the only kind of data it can hold. This is not true of PAL variables, however. When you create a variable by assigning a value to it, the variable takes on the type of that value, but only while storing that value. Thus, these statements are perfectly valid in a single PAL script:

```
current = 10/12/45
current = 10.3
current = "Nope, not yet..."
```

PAL’s automatic typing means that you do not have to bother with the type declaration statements required in other programming languages. However, you should make sure that the information you’re manipulating is in the form you want it to be. This is especially important since PAL performs automatic type conversion in many of its operations (see “Data types” in Chapter 3). For example, if you add a numeric value and a currency value together, you get a currency value that is the sum of the two.

Many PAL commands and functions only work when you pass them data of a specific type. The ACOS() (arc cosine) function, for instance, will produce a script error if you pass it string “two”. How do you ensure that you’re working with the right type of data if the variable automatically follows the type of the data assigned to it? There are two ways to prevent a data type error:

- Use the TYPE() function to explicitly check the type of the variable before passing it to the function. TYPE() returns a string telling you what the data type of an expression is:

```
IF TYPE(current) = "D"
  THEN ? "Current is currently a date value"
  ELSE ? "Current is not a date"
ENDIF
```

- Make sure that the variable never contains anything but a number in the first place. One method of facilitating this is to name all variables in a predictable pattern like

*Avarname* for alphanumeric (string) variables

*Nvarname* for numeric variables

*Dvarname* for date variables

*Cvarname* for currency (\$) variables

Of course, naming your variables this way won't keep Paradox from converting variables automatically and placing a date value in a logical variable, but it helps you verify scripts. If you see a date variable being assigned a value, you can verify that it is a date value. Likewise, you'd know something was wrong if you saw:

```
ACOS(DateVarname) ; trigonometric function on a date?
```

The important point is not that you name variables exactly as described here, but that you use a consistent system that lets you quickly and correctly identify what type of data the variable contains.

---

## Using fixed arrays

If a variable is like a mailbox, arrays{ a fixed array is like a whole row of mailboxes for an office or apartment building. Unlike variables, you must declare and dimension fixed arrays. For instance:

```
ARRAY a[5] ; creates an array called a with five elements
a[1] = 5 ; bracketed subscripts to refer to each element
a[2] = 3
a[3] = a[1] * a[2]
a[4] = 10/12/89
a[5] = "Now is the time"
a[2] = "for all good men and women"
a[2] = a[3] + x
[Orders -> Stock #] = a[4]
```

As you can see, you can mix data types within elements of the same array, and reassign an element of one type as another.

---

## Manipulating records in fixed arrays

One of the most useful applications of fixed arrays is to store the field values of an entire record in a Paradox table. You can use arrays in this way to manipulate the contents of a record and to move records around a table or from one table to another.

Three special PAL commands, COPYTOARRAY, COPYFROMARRAY, and APPENDARRAY use fixed arrays to work with records. These commands assume that the current image on the workspace is a display image in either table or form view. COPYTOARRAY and

COPYFROMARRAY are available in Edit or CoEdit mode; APPENDARRAY assumes that the workspace is in CoEdit mode. COPYTOARRAY creates (declares and dimensions) a new array automatically and copies the fields of the current record to it, while COPYFROMARRAY transfers the values stored in an existing array to the current record. APPENDARRAY appends records in a comma-separated list of fixed arrays to the end of a table. You cannot use COPYTOARRAY, COPYFROMARRAY, or APPENDARRAY with dynamic arrays.

For example, the following script copies the values in the current record of a table to a record in another table:

```
COPYTOARRAY TransferRec      ; create array TransferRec, store record in it
MOVE TO "BackUp"            ; move to the BackUp table
COEDITKEY                    ; turn on CoEdit mode for transfer
INS                           ; press insert key to insert record
COPYFROMARRAY TransferRec    ; transfer the stored record
DO IT!                       ; end CoEdit mode
```

Since COPYTOARRAY automatically creates the array named as its argument, no explicit declaration of the array is required when you use it. The number of elements in the array will correspond to the number of fields in the current record plus 1. The elements are assigned in this way:

- The first element contains the name of the table.
- The remaining elements contain the values of each of the fields of the current record, in the order the fields appear in the table.

Once you've used COPYTOARRAY, you can reference each element in the array in the normal way. For instance, if you just grabbed a record with COPYTOARRAY and put it into an array called *reccopy*, to get the name of the table you could use

```
TblName = reccopy[1] ; assign table name to TblName
```

You can also manipulate the contents of that array before using COPYFROMARRAY to write it back to a table. (However, in a multi-table form, you can't manipulate the key field in the array, then use COPYFROMARRAY to copy it into a detail record.) See Example 19-1 for an illustration.

APPENDARRAY appends the elements of each fixed array in a comma-separated list as a new record in the current image. If the table is keyed, each record is inserted in the proper index order. If the table is not keyed, each record is appended to the end of the table.

The arrays are assumed to have been created by COPYTOARRAY or to have the form of such an array; that is, the first element of the array is assumed to contain the table name and is therefore ignored. The transfer of field values begins with the second element of the array, which is copied to the first field of the current record. The third

---

## Field names as subscripts

element of the array is copied to the second field, and so on until either the array or the record is exhausted.

In a multi-user environment, APPENDARRAY is faster and more convenient than issuing several COPYFROMARRAY commands. See the *PAL Reference* for examples using APPENDARRAY.

In an array created by COPYTOARRAY, you can access particular fields by name as well as by number. For example, if the record stored in array *rec* is a record of the *Orders* table, then the expression

```
rec["Stock #"]
```

refers to the value of the Stock # field in the record stored in the array. The reference to the field name must be a string, either an explicit string (as above) or the result of an expression that evaluates to a string. Remember that the array element for a field is the field number plus 1. For example, since Stock # is the third field in *Orders*, the references `rec["Stock #"]` and `rec[4]` are equivalent after COPYTOARRAY is executed.

See the descriptions of COPYTOARRAY and COPYFROMARRAY in the *PAL Reference* for more details on using these commands.

---

### Example 19-1 Manipulating array elements

---

Here is a *Reorder* procedure that copies an *Orders* record to array *b*, assigns a new Order #, changes the order date to today, and writes the record back to *Orders*.

You could incorporate this procedure in a keyboard macro so that a sales representative could enter a reorder with a single keypress.

```
PROC Reorder()
  IF TABLE() <> "Orders"                ; make sure Orders table is current
    THEN MESSAGE "Must be in Orders table"
      SLEEP 1500
      MESSAGE ""
      RETURN False                        ; return False to calling script
  ENDIF
  COEDITKEY                               ; enter CoEdit mode
  COPYTOARRAY b                            ; pick up current record
  b["Order #"] = CMAX("Orders", "Order #") + 1 ; assigns next order number
  b["Date"] = TODAY()                      ; assign today's date
  END                                       ; go to end of table
  DOWN                                     ; move down to blank record
  COPYFROMARRAY b                          ; copy to blank record
  DO_IT!                                   ; end CoEdit session
ENDPROC
```

## Using dynamic arrays

Because the index of a dynamic array is not just limited to integers, you can use a dynamic array to store and manipulate a broad range of value relationships.

When you use fixed arrays, each element associates an index which is an ordinal number with a value. Dynamic arrays let you associate two *values* of any type instead of only a value and an ordinal index. For example, if you use a dynamic array to manipulate table values, each element can associate the values in two separate fields.

Like elements in fixed arrays, elements in dynamic arrays can hold almost any value, including the contents of memo fields and the concatenation of fields.

### Example 19-2 Manipulating dynamic array elements

You can use a dynamic array to hold the contents of the Last Name and Date Hired fields from the *Employee* table shown here.

EMPLOYEE	ID #	Last Name	Employee Init	Position	Date Hired
1	146	Christiansen	S	Sales Rep	1/25/83
2	395	Chestnut	R	Dept Mgr	7/12/76
3	422	Kling	W	Sales Rep	3/31/85
4	517	Morris	T	Telephone Sales	4/09/81
5	537	Lee	Y	Secretary	12/01/82
6	775	Chambers	M	Sales Rep	7/14/78
7	900	Jones	L	Admin Asst	3/05/80

The script used to create such an array is:

```
DYNARRAY Seniority[ ]
SCAN
  Seniority[[Last Name]] = [Date Hired]
ENDSCAN
```

The elements of this dynamic array would look like this:

```
SENIORITY["KLING"] = 3/31/85
SENIORITY["CHRISTIANSEN"] = 1/25/83
SENIORITY["CHAMBERS"] = 7/14/78
SENIORITY["JONES"] = 3/05/80
SENIORITY["LEE"] = 12/01/82
SENIORITY["CHESTNUT"] = 7/12/76
SENIORITY["MORRIS"] = 4/09/81
```

You can use a dynamic array created in this manner as a high speed alternative when looking up table values.

### Example 19-3 Simulating object-oriented programming

You can also take advantage of dynamic arrays to simulate "object oriented" programming. For example, you can create windows and then use a dynamic array to associate the window handles with procedure names. When a user

clicks on one of these windows, the procedure that is associated with that window is automatically executed—like a method that is associated with an object in an object-oriented language.

The code needed to create this effect could look like the following:

```

PROC CustomerViewer()                ; this procedure views Customer
  ECHO OFF
  VIEW "\\PDOX40\\SAMPLE\\CUSTOMER"
  ECHO NORMAL
ENDPROC

PROC BookordViewer()                 ; this procedure views Bookord
  ECHO OFF
  VIEW "\\PDOX40\\SAMPLE\\BOOKORD"
  ECHO NORMAL
ENDPROC

DYNARRAY WindowObject[]             ; create a dynamic array
WindowObject["HASFRAME"] = False    ; that contains
WindowObject["HASSHADOW"] = False
WindowObject["WIDTH"] = 16
WindowObject["HEIGHT"] = 1
WindowObject["FLOATING"] = True

WINDOW CREATE ATTRIBUTES WindowObject @23,0 TO CustomerWin
? " View Customer "                  ; clicking this window
                                     ; views the Customer table

WINDOW CREATE ATTRIBUTES WindowObject @23,16 TO BookordWin
? " View Bookord "                   ; clicking this window
                                     ; views the Bookord table

WindowObject[CustomerWin] = "CustomerViewer" ; add additional elements to
WindowObject[BookordWin] = "BookordViewer"   ; the dynamic array- the
                                               ; elements correspond to
                                               ; window handles (no "")

WHILE True
  GETEVENT MOUSE "DOWN" TO EventArray      ; trap only mouse DOWN events
  ; now assign the ProcName variable with the handle of the
  ; canvas window that the mouse click occurred in
  ProcName = WINDOWAT(EventArray["ROW"], EventArray["COL"])

  IF (ProcName = CustomerWin) OR           ; if user clicks on one of
    (ProcName = BookordWin)                ; the canvas windows,
  THEN EXECPROC WindowObject[ProcName]    ; execute the procedure
  ELSE EXECEVENT EventArray                ; otherwise execute the event
ENDIF
ENDWHILE

```

In this example, two procedures are defined and two floating windows are created. The dynamic array *WindowObject* contains tags that are window handles and element values that are procedure names. The `GETEVENT` statement traps mouse `DOWN` events and evaluates them. If the mouse down event occurs within a window, the procedure associated with that window is executed. If the mouse `DOWN` event occurs outside a window, the event itself is executed.

---

## Using variables to compute commands

You can use variables and arrays with the EXECUTE, KEYPRESS, TYPEIN, and SELECT commands to compute commands and simulated keypress interaction at runtime. For example, suppose only the ← and → keys apply to a certain situation.

```
Char = GETCHAR() ; get keystroke from the user
SWITCH
  CASE Char = ASC("Right"): KEYPRESS Char
  CASE Char = ASC("Left") : KEYPRESS Char
  OTHERWISE                : BEEP
ENDSWITCH
```

The above example intercepts the user's keystrokes and passes through to Paradox only ← and → as actions. Since GETCHAR() doesn't actually execute the character, the first line simply stores it in the variable *Char*, giving you a chance to see what the user is doing before allowing it to happen. In this instance, you only want the user to move the cursor, so you use KEYPRESS to simulate the typing of those characters as stored in *Char*. If the user types something else, the only response is a beep.

You could use a similar procedure to fill in data automatically. For example,

```
Char = GETCHAR() ; get keystroke from user
SWITCH
  CASE Char = ASC("A") : TYPEIN "Automatic"
  CASE Char = ASC("a") : TYPEIN "Automatic"
  CASE Char = ASC("M") : TYPEIN "Manual"
  CASE Char = ASC("m") : TYPEIN "Manual"
  OTHERWISE            : TYPEIN "Not specified"
ENDSWITCH
```

The above example fills in a complete word when the user types a single character. The effect is similar to using an alternative picture with autofill (see Chapter 5) or the HelpAndFill option of a table lookup (see Chapter 11 of the *User's Guide*).

You can use string functions and operators with these commands to combine variables with explicit strings. For example, this EXECUTE command assigns the value 45 to the *Fld* field in the current record of the *TabName* table:

Variable

```
Fld = "Stock #"
TabName = "Products"
EXECUTE "[" + TabName + "->" + Fld + "] = 45"
```

```
graph TD
    Strings --- P1["+"]
    Strings --- P2["+"]
    Strings --- P3["+"]
```

The + signs in the above example are used to concatenate the separate elements into a single string expression. The string expression that you supply with the EXECUTE command in this manner is limited to 174 characters.



You can use EXECUTE with a variable in places where PAL expects literal characters instead of a string. For example, when you place a write lock on a table, the PAL LOCK command expects the literal characters *WL*, not the string "WL". If you assign the value *WL* to a variable and try to use the LOCK command without EXECUTE, the command will fail because PAL interprets the value as a string:

```
; this LOCK command will fail!
TabName = "Products"
LockType = "WL"
LOCK TabName LockType ; fails
```

Using EXECUTE with the LOCK command here will allow the command to complete successfully:

```
; this LOCK command will succeed!
TabName = "Products"
LockType = "WL"
EXECUTE "LOCK " + TabName + " " + LockType ; succeeds
```

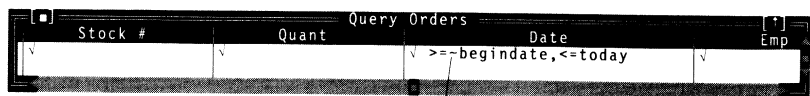
## Query variables

You can use variables in query statements by preceding them with a tilde (~) character. When Paradox encounters a tilde variable in a query statement, it replaces it with the current value of the variable. For example, these commands ask the user for a date in the past and then display all orders placed since that date:

```
? "What's the earliest order date you want to see? "
ACCEPT "D" MAX TODAY() TO begindate
PLAY "Recent" DO IT!
```

where *Recent* is a query script for the query shown in Figure 19-1.

Figure 19-1 Variables in queries



Variable identified by tilde

You can use query variables to store almost any item that can be entered in a field of a query form, as described in Chapter 5 of the *User's Guide*. There are several special cases:

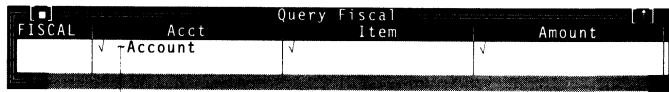
- Precede the text of an example element with an underbar (\_). For example, type *\_cust* to represent the example element *cust*.
- Do not use an underscore as part of the name of a query variable because the part of the variable that follows the underscore will be treated as an example element.

- ❑ Use the keywords CHECK, CHECKPLUS, CHECKDESCENDING, and GROUPBY to represent ✓, ✓+, ✓▼, and G (Groupby).
- ❑ Use a backslash (\) to precede special characters, like quotation marks, in strings.
- ❑ Use variables to represent literal values only, not wildcards or query keywords. For example, you can use a wildcard operator (.. or @) as part of the query statement but not part of the variable assignment. To find all order numbers ending in 301, assign "301" to the *Ordnum* variable. Then type **..~ordnum** in the Order # field of the query statement.

Query variables are also handy when you have two or more similar queries that you use with the same table. Instead of creating several different query scripts, you could create just one and use a query variable to alter it.

### Example 19-4 Using variables to adjust a query

Suppose you want to display records for one accounting category at a time. Create a query to retrieve the line items for whichever account number is stored in the variable *Account*. Save it in a script called *Showacct*.



FISCAL	Acct	Item	Amount
	Account		

Tilde variable

Use `SHOWARRAY` to let the user choose which account to display. Remember to use the `DO_IT!` command to execute the query.

```
ARRAY Category[10]
Category[1] = "Cash"
Category[2] = "Accounts Receivable"
Category[3] = "Short-term investments"
Category[4] = "Fixed Assets"
Category[5] = "Salaries"
Category[6] = "Rent"
Category[7] = "Utilities"
Category[8] = "Cost of Goods"
Category[9] = "Shareholder Equity"
Category[10] = "Retained Earnings"
```

```
ARRAY ItemNumber[10]
ItemNumber[1] = "012"
ItemNumber[2] = "024"
ItemNumber[3] = "053"
ItemNumber[4] = "144"
ItemNumber[5] = "222"
ItemNumber[6] = "246"
ItemNumber[7] = "286"
ItemNumber[8] = "332"
ItemNumber[9] = "690"
ItemNumber[10] = "683"
```

```
SHOWARRAY Category ItemNumber
TO Account
```

```
PLAY "Showacct"
DO_IT!
```



# Keyboard macros

This chapter shows you how to define and use keyboard macros to streamline the process of calling scripts down to a single keystroke. You can use keyboard macros to make yourself or your users more productive, and to customize Paradox to individual tasks and situations.

A *keyboard macro* is a single key or keystroke combination that causes Paradox to execute a specified series of keystrokes or commands. For instance, you might want to be able to press *Alt-Q* to perform a query on particular table. This chapter explains how to

- define keyboard macros
- use keyboard macros to
  - update records
  - sort and reorder tables
  - present information in different ways
- cancel keyboard macros

---

## Defining macros with SETKEY

Use the SETKEY command, described in detail in the *PAL Reference*, to create keyboard macros. SETKEY takes a series of commands and tells Paradox to play them whenever a particular key is pressed. The specific form of the command is

```
SETKEY Keycode Commands
```

where *Keycode* is the keycode representation of the key that starts (invokes) the macro, and *Commands* is the series of commands to execute when that key is pressed.

A SETKEY macro must be valid in the mode in which it is used. For example, a macro that sorts a table must be used in Main, Edit, or CoEdit mode; if you attempt to use it during an Editor session, it will cause a script error.

The best way to execute SETKEY commands depends on how you want to use the macros:

- To define macros on the fly, use MiniScript from the PAL or Debugger menu *Alt-F10*.
- To capture a sequence of keystrokes in a macro, use Instant Script Record *Alt-F3*. Then rename the resulting *Instant* script and use MiniScript to attach it to a macro.
- To use the same set of macros all the time, put the SETKEY commands in your *Init* script so the macros will always load with Paradox.

---

## Keycodes to use

Recall from “PAL keycodes” in Chapter 3 that Paradox keycodes can be represented in several ways:

- as a single-character string (such as “a”, “Z”)
- as a numeric value (45, 255, -128)
- as one of the specially recognized key strings (“F10”, “Right”, “Do\_It!”, “F2”)

Thus, you have the ability to assign commands to just about any useful key or combination of keys on the keyboard.

Obviously, it doesn’t make much sense to assign macros to commonly used keys, such as letters, numbers, arrow keys, or *Space*. Also, it’s not advisable to assign macros to important Paradox keys, such as *Enter*, *←*, *→*, *Do\_It!*, *F2*, and so on. If you do so, you could create an endless loop—a situation the user can’t get out of because a crucial key has been redefined. However, you might want to reassign rarely used keys with the SETKEY command.

Appendix G of the *PAL Reference* contains a complete list of all possible macro keycodes.

---

## Commands to use

Following the keycode, you can list any number of PAL commands to be executed when the key is pressed. For example,

```
SETKEY -33 "Luxury Gifts Department"
```

types in **Luxury Gifts Department** when the user presses *Alt-F* just as if the user typed it. Similarly,

```
SETKEY -20 TYPEIN TODAY()
```

types in today's date when the user presses *Alt-T*, while

```
SETKEY "/" Menu
```

makes the current menu active when the user types a slash, as the slash in QUATTRO PRO does. (However, the user won't be able to enter expressions that use division because of this reassignment.)

The only restriction is that the commands must fit on the same line as the SETKEY command. If the commands won't fit on one line, put them in a script and use a PLAY command to reference them in the macro. This is perhaps the most powerful method for defining keyboard macros because you can activate any arbitrary script or even an entire application by pressing a single keystroke. Suppose you want to press *Alt-Q* to perform a particular query. Since a complete QUERY statement takes several lines, use Scripts | QuerySave to save the query statement in a script. Then use

```
SETKEY -16 PLAY "Query1" DO_IT!
```

to attach the query to *Alt-Q*. The DO\_IT! command at the end of SETKEY executes the query stored in the *Query1* script.

---

## Using macros

Now that you know how to create a macro, let's examine some ways you can use them. The primary advantage of keyboard macros is the productivity enhancement they bring. If you consider the things you as an application developer might want to do with a single keystroke, they would probably fall broadly into two categories:

- streamline a user's interaction with an application
- increase your productivity in programming and debugging applications

### Example 20-1 Using macros to update records

---

Suppose you're developing an application for a magazine publisher. In a table of subscriber information, there is an expiration field (Expr) that indicates the date each subscription expires. Subscription renewals can be for one, two, or three years. You would like subscription personnel to be able to pull up the database, find the right subscriber, and update the expiration date with a single keystroke. You'll need three different keyboard macros for the updates.

```
SETKEY -120 [Expr] = [Expr] + 365 ; 1-year renewal (Alt-1)
SETKEY -121 [Expr] = [Expr] + 730 ; 2-year renewal (Alt-2)
SETKEY -122 [Expr] = [Expr] + 1095 ; 3-year renewal (Alt-3)
```

When you press *Alt-1*, *Alt-2*, or *Alt-3*, the expiration field in the current record will be extended by the appropriate number of years. The table must be in Edit or Coedit mode when any of the macro keys are pressed.

Up to the number of keys that Paradox understands, there is no limit to the number of macros you create. We suggest making your keyboard macros memorable by using mnemonics. For example, *Alt-1* reminds the user that the update is for one year.

---

## Using macros to reorder tables

You can define macros to help you quickly change the order of records in a table. For example, the following macro lets you move records around in a table:

```
SETKEY -46 COPYTOARRAY A De1 ; Alt-C "Cut" macro
SETKEY -25 Ins COPYFROMARRAY A ; Alt-P "Paste" macro
```

To move a record to another location, make sure you're in Edit or Coedit mode, move the cursor to the record, and press *Alt-C*. The Cut macro uses the COPYTOARRAY command to copy the current record to array *A*, then deletes it from the table image. Then move the cursor to the place you want the record and press *Alt-P*. The Paste macro opens up a blank record, then uses COPYFROMARRAY to copy the record from array *A* back into the table. (See the description of COPYFROMARRAY in the *PAL Reference* for important notes on using it with a multi-table form.)

---

## Using macros to present information

Paradox provides a variety of ways in which it can display data for you. You can look at information formatted as a table, as a form, or even as a report.

Creative use of macros to present information onscreen helps you focus on just the information you need, while still letting you access the full set of data.

### Example 20-2 Presentation macros

---

Suppose a magazine publisher's table of subscriber information contained fields for subscriber name, title, company, address, city, state, zip code, expiration date, and other data. In table view, all this information won't fit on the screen at once. In order to see it all, you can use form view, thus restricting users to one record at a time.

You could have two forms for subscriber information:

- one that shows only the most common information (like subscriber name, address, and expiration date)
- a second that contains detailed information, including initial subscription date, payment method, priority airmail orders, and so on

You could then define keyboard macros to switch back and forth between the two views:

```
SETKEY "F21" PICKFORM 1
; Ctrl-F1 for important info
SETKEY "F22" PICKFORM 2
; Ctrl-F2 for complete info
```



If you want to see the condensed form view, press *Ctrl-F1*. To see the entire record of information, press *Ctrl-F2*.

---

## Using macros for convenience

As described in Chapter 9, you can use the Custom Configuration Program to link an external program editor into the Paradox environment. However, you can leave the PAL Editor as your primary editor and still have access to an external editor by creating a keyboard macro for the external editor. For example, if you want to edit scripts with Sprint, the macro

```
SETKEY 19 RUN "SP" ; sets Ctrl-S to Sprint
```

suspends Paradox and loads Sprint whenever you press *Ctrl-S*.

You could also use a macro to help you avoid accidentally deleting a record from a table. For example, the following script uses SETKEY to display a pop-up menu for confirmation before deleting a record, checking to make sure that Paradox is in Edit or Coedit mode and not in field view. This keyboard macro could go in a user's *Init* script:

```
SETKEY -83 PLAY "SafeDel"

; The SafeDel script confirms the Del key before deleting a record
; in Edit or CoEdit mode.

IF (SEARCH("Edit", SYSMODE()) > 0) AND ; if we're in Edit or CoEdit mode
NOT ISFIELDVIEW() ; and not in fieldview
THEN ; display a pop-up menu
SHOWPOPOP "Are you sure?" CENTERED
"No" : "Do not delete the current record" : "Confirm/No",
"Yes" : "Delete the current record" : "Confirm/Yes"
ENDMENU TO Choice
IF Choice = "Confirm/Yes"
THEN DEL
ENDIF
ELSE DEL ; if we're not in Edit or Coedit
ENDIF ; press the Del key
```

---

## Using macros wisely

A few cautions should be mentioned in connection with keyboard macros. One has been discussed: It is not a good idea to use SETKEYs to redefine commonly used Paradox keys. For example, if you redefined *F10* as Edit, most users would not be able to work very well with Paradox.

Also, if you use a SETKEY that expects a certain starting environment (for instance, a table on the workspace), SETKEY won't work unless you're in that environment when you play it.

Another potentially troubling aspect of macros is when Paradox behaves in unusual ways with no apparent explanation. For example, instead of getting a report when Instant Report *Alt-F7* is pressed, a user's screen might be cleared. Or, *Del* no longer works when a user tries to delete a record. In such situations, hidden or forgotten keyboard macros are often the culprit. Check for macros that you

have put into an *Init* script and forgotten about. See the next section for further suggestions on how to remedy such a situation.

---

## Canceling macros

Once you issue a SETKEY command, your keyboard macro remains in effect for the duration of the Paradox session, or until you use SETKEY again to redefine the same key. You can cancel a keyboard macro by issuing SETKEY with its *Keycode* but no *Commands*. For example,

```
SETKEY "F21"  
SETKEY "F22"
```

cancels the *Ctrl-F1* and *Ctrl-F2* Pickform macros.

In general, you should cancel macros when you are done using them. This prevents macros from being used in the wrong context. To cancel all macros, you could run this script:

```
FOR i FROM 0 TO 255           ; clear ASCII SETKEYs  
  SETKEY i  
ENDFOR  
  
FOR i FROM -2 TO -132 STEP -1 ; clear extended character SETKEYs  
  SETKEY i  
ENDFOR
```

# Performance and resource tuning

This chapter explains how to design your application to run at maximum speed on a variety of machines. Essentially, there are two things you can do to speed up your applications:

- Design or modify your program to use Paradox's automatic memory management features and your machine's memory resources efficiently.
- Reconfigure, optimize, or increase the resources of the machine running the application.

This chapter also provides detailed information about how PAL manages memory, and how to fine-tune Paradox's default settings to optimize the performance of your application. Chapter 23 of the *User's Guide* describes how Paradox manages memory for different machine configurations. If you are unfamiliar with the terms *extended memory*, *expanded memory*, and *protected mode*, you should read about memory management in Chapter 23 of the *User's Guide* first.

---

## Scripts

The first time a script is executed and every time a script is changed, Paradox loads the script into memory, parses it, and saves this "pre-parsed" version of the script on disk as an .SC2 file. After the .SC2 file is created, a script runs faster because Paradox can use this pre-parsed version of the script. Paradox can swap procedures loaded from a .SC2 file in the same manner that it swaps procedures loaded from a library.

Regular scripts have the file extension .SC; pre-parsed scripts are given a .SC2 extension by Paradox. You should include a .SC2 file with any .SC file that you distribute in your applications. If you do not distribute the .SC2 files, Paradox will simply regenerate them the first time your application runs; however, your application will run

faster that first time if Paradox does not have to pre-parse your scripts. If your distributed application is installed in a write-protected directory, it is more important to include the .SC2 files. If you install your application in a write-protected directory without providing the .SC2 files, Paradox must keep the application in memory. Installing an application in a write-protected directory without .SC2 files results in a significant performance penalty; this is the only way to prevent Paradox from automatically swapping procedures.

Until your application is distributed, you should always keep your scripts in directories that are not write-protected so Paradox can continue to create pre-parsed files as you continue to revise your scripts.

Because procedure libraries now contain the same pre-parsed procedures that are available on disk as .SC2 files, there is no longer a performance advantage in creating procedure libraries (as there was in versions of Paradox prior to 4.0). There are, however, many other reasons to continue to create procedure libraries. These reasons are described in Chapter 6.

Because Paradox pre-parses a procedure in memory and then creates a .SC2 file, you should avoid using excessively large procedures. You obtain a significant performance advantage from pre-parsed scripts, but you don't want to run out of memory trying to create them!

---

## Procedure and script swapping

In Paradox 4.0, all scripts and procedures except the one currently executing are fully swappable; restrictions on swapping in previous versions of Paradox no longer apply. Once a procedure is read or automatically loaded from a library, Paradox swaps it out of and back into memory as needed. This means that your application can call an almost unlimited number of procedures without running out of memory.

When Paradox needs a procedure, it looks for the procedure in the following places in this order:

1. Paradox first looks for the procedure in memory.
2. Paradox then looks for the procedure in the *cache* (a storage area in memory). A procedure might be in the cache if the application has called it once already.
3. Paradox next searches on disk in the library the procedure was originally read from, even if that library is no longer on the current autoload library path.

4. Finally, Paradox looks on disk along the current autoload library path, as specified with the *Autolib* system variable.

If Paradox finds the procedure in memory, it simply executes it. If Paradox finds the procedure in the cache, it copies the procedure to another part of memory and executes it from there (not from the cache). If Paradox finds the procedure on disk, it copies the procedure to the cache *and* to memory. Paradox copies procedures from disk to the cache so the next time the procedure is called, it can be retrieved from the cache. Because the cache resides in RAM, future reads from the cache are faster than reads from disk.

Paradox swaps out procedures so that it can make room to load the next procedure or perform the next operation. This ability to dynamically free memory is what keeps your application running faster because it keeps your application from running out of memory. It follows, then, that you should structure your application so that Paradox can swap out procedures.

The swapping mechanism works best if procedures are small. As a rule of thumb, procedures should be less than 10K. Smaller procedures also allow a larger working set of procedures to stay in memory at the same time.

Paradox can swap out all called procedures except the one it is currently executing. In other words, Paradox cannot swap the current procedure on the *call chain*. It can, however, swap procedures higher on the call chain than the current procedure. (That is, when *ProcName1* is executing, it is the current procedure on the call chain. When *ProcName1* calls *ProcName2*, *ProcName2* becomes the current procedure, and *ProcName1* is higher on the call chain than *ProcName2*.)

For example, if *ProcName1* calls *ProcName2*, then Paradox can swap out *ProcName1* before it loads and executes *ProcName2*. When *ProcName2* ends, Paradox can swap out *ProcName2*, and swap *ProcName1* back in.

---

## Managing memory in interactive Paradox

Paradox prefers extended memory to expanded memory and will try to configure as much memory as possible as extended memory. Paradox can use all available extended memory to load the Paradox program, tables, procedures, pre-parsed scripts, and objects. In addition, Paradox allocates extended memory for a cache.

Paradox does not use extended memory as a swap device. Instead, Paradox directly manipulates an object from its location in extended memory. When Paradox needs to make more room in memory, it swaps objects to and from an extended memory cache. If you have

enough extended memory to hold all of the procedure libraries for an application, you can eliminate the need for disk swapping altogether. Depending on how much extended memory is available, Paradox can potentially deliver much more working space for your application with extended memory than it can with an expanded memory swap device.

---

## Swap devices

Expanded memory, unlike extended memory, will be configured as a swap device by Paradox. In other words, Paradox will load an object in regular memory, then swap it to and from an area in expanded memory. It is preferable to have extended memory; however, Paradox can still use any expanded memory as a high-speed storage device.

If you have a swap device, Paradox uses part of it to hold most or all of the cache. Among other things, the cache stores procedures recently called from libraries and pre-parsed scripts from .SC2 files. Procedures from PAL libraries and pre-parsed scripts are not directly executed from their location in a cache. However, when Paradox needs to swap a procedure or pre-parsed script out of regular memory, it discards it to make room for another memory consumer. When Paradox needs the procedure or pre-parsed script again, it can be read from the cache much more quickly than from disk. (This process works for designated error procedures as well.)

---

## Command-line options

The “Command-line configuration” section in Chapter 23 of the *User’s Guide* lists command-line options for such things as how much extended or expanded memory to use as a swap device, how much memory to allocate for the internal stack, and so on. If you know the specific resources of the machine on which your application will run, you might want to write a batch file that invokes Paradox and your application with any command-line options you think will increase performance.

---

## Managing memory in an application

Using VROOMM (the Virtual Runtime Object-Oriented Memory Manager), Paradox balances the needs of a particular memory consumer, like the Paradox program itself, against the needs of other consumers, like images, procedures, variables, and others. In different processing situations, the needs of these consumers change, and Paradox adjusts to those needs.

Consider what kind of machine or machines an application will run on. If an application runs on only one type of machine, you can set up memory management to suit the needs of that particular machine configuration. If the application must run on a variety of machines

(as Paradox itself does), then you'll need to build flexibility into your memory management scheme.

Paradox allocates memory to an area called the *central memory pool*. The central memory pool resides in both regular and extended memory with other memory consumers. The size of the central memory pool varies depending on how much memory is available in the machine and on Paradox's current operation. During a sort, for instance, where another memory area needs to grow very large, the central memory pool shrinks.

Part of the central memory pool is set aside for a special area called a *cache*. The cache stores a copy of the procedures, pre-parsed scripts, tables, form specifications, and other Paradox objects that have most recently been read from disk. When Paradox needs the procedure or object again, it looks in the cache. If Paradox finds the procedure or object in the cache, Paradox can transfer it to regular memory much more quickly than if Paradox had to read it from disk.

In earlier versions of Paradox, the **-cachek** command-line option was used to control the size of the cache. Starting with Paradox 4.0, you should only use the **-cachek** command-line option in limited memory situations (machines with 1-2 megabytes of RAM); in other cases, Paradox will adjust the size of the cache for you. See Chapter 23 of the *User's Guide* for details about the **-cachek** command-line option.

You can interactively tune Paradox in low-memory situations by using the **-space** option to determine your current settings and adjusting the **-tablek** and **-codepool** command line options. See Chapter 23 of the *User's Guide* for details about starting Paradox with these command-line options.

---

## **RMEMLEFT() and the code pool**

Besides the central memory pool, there is another area of reserved memory called the *code pool*. The code pool is an area of memory that Paradox saves to store its own program code. When the central memory pool runs out of memory, Paradox can use some of the code pool's memory. However, a low memory error (error 44) occurs when Paradox uses too much of the code pool's memory. Because Paradox needs the code pool to run efficiently, you generally do not want your application to invade the code pool.

There are two parts to the code pool. You can use the upper part of the code pool without causing an error; Paradox lets you borrow this area. (Paradox runs slower if you use the upper part of the code pool, but this is not a serious condition.) If however, you cross the boundary into the lower part of the code pool, you get error 44 (low memory error). This error indicates a serious condition that should be corrected immediately.

**Note** Error 44 only occurs when the application *crosses* from the upper area of the code pool into the lower area of the code pool. Continuing to invade the lower area doesn't trigger successive error 44s. This makes it possible to use a little more memory while you're taking actions to back out of the code pool.

To find out how much memory is available in the code pool, use the RMEMLEFT() function. The checking account analogy might be helpful again. You can think of the code pool as overdraft protection. RMEMLEFT() tells you how much memory you have left to spend before your application stops.

RMEMLEFT() sets the error code to 44 if the lower part of the code pool is being used by an application. If only the upper part of the code pool is being used by an application, the ERRORCODE() function returns 0 (zero). It is useful to test the error code with the ERRORCODE() function as you're backing out of the code pool after a low memory error. If ERRORCODE() continues to return 44, this means memory is not being freed, and therefore you should end the module or application. Continuing to invade the code pool after a low memory error causes unpredictable results; eventually, your application stops.

---

### **Handling a code pool invasion**

If you suspect your application will cause a low memory error, you should write an error procedure to handle it. (See Chapter 7 for general information on error procedures and the special system variable *Errorproc*.)

If an error other than error 44 occurs, and while handling that error your application crosses from the upper part of the code pool into the lower part, this action also triggers error 44. It follows then that you should test MEMLEFT() or RMEMLEFT() before loading an error procedure that uses more memory.

There are essentially two methods for storing and calling error procedures: You can define the entire error procedure at the beginning of the application, or you can call an error procedure from a regular or autoloading library.

If you define the error procedure at the beginning of the application, the procedure is loaded into memory and stays there until it is specifically released or the program terminates. If the procedure is large, this is not a desirable method because it uses too much memory, but this method is effective for handling an error 44.

To use memory more efficiently, you can define an error procedure, store it in a regular library, and explicitly read it in with the READLIB command at the beginning of your application. A better technique is to store the error procedure in an autoloading library so that Paradox reads it from the autoloading library as needed.



---

## Indexing tables

You can dramatically improve the performance of queries, ZOOM, and LOCATE commands in your applications by indexing tables used by the application. Primary indexes are automatically built and maintained on keyed tables. They provide nearly instantaneous access to records that can be identified by their key values. In addition, you can use the INDEX command to build secondary indexes. For application developers, using keyed tables and maintained indexes is almost always a sure way to speed processing. However, the advantage of faster queries and searches is offset to some degree by the time it takes Paradox to maintain the index. If your application rarely performs queries and searches, then it would be pointless to slow other operations for the sake of index maintenance. For a general discussion of indexes, see “Indexing tables” in Chapter 23 of the *User’s Guide*.

Paradox automatically utilizes indexes whenever they are present; there’s no need to explicitly open or use an index. In addition, indexes are strictly optional. For these reasons it’s possible to ignore indexes while you are developing an application. Once the application is working, you can put them in place to optimize performance.

In addition to using the INDEX command, there are other ways that secondary indexes can be built for a table:

- ❑ You can interactively build an index for a table by using Menu {Modify} {Index}.
- ❑ Paradox sometimes automatically builds a secondary index during query execution if it determines that this is the most efficient way to process the query; however, these indexes are temporary.
- ❑ You can optimize a query for an application by placing the query on the workspace and using Menu {Tools} {QuerySpeed} to build the secondary index interactively.

When you use QuerySpeed, Paradox automatically builds a secondary index for each field in the current query statement for which there are selection criteria specified (if the field search can benefit by an index). Because this method can sometimes create unneeded indexes, using the INDEX command is generally the preferred approach as it lets you select exactly which fields to index. This lets you optimize the sometimes delicate balance between query speed and the disk space required for the secondary indexes. There is never any reason to build secondary indexes on the first key field of a keyed table.

Once an index is created, Paradox updates and uses it automatically as described in the next section, "How indexes are maintained." You can also update an index by using INDEX to regenerate it.

#### Example 21-1 Indexing a table

---

Suppose that queries of the sample *Products* table often select records based on the Description and Price fields. To create a secondary index on Description, you could add this command to the section of the application that builds the tables:

```
INDEX MAINTAINED "Products" ON "Description"
```

---

## How indexes are maintained

Secondary indexes can be maintained in batch mode or incrementally. In batch mode, the index is not updated when you complete a change to a table; rather, it is marked obsolete and is completely regenerated the next time it is about to be used. Depending upon the size of the table and the number of indexes involved, regenerating the indexes on a table could take a significant amount of time relative to query processing.

All secondary indexes on unkeyed tables are maintained in batch mode, as are indexes on keyed tables created without the INDEX command's MAINTAINED option or without the CCP MaintainIndexes command.

If the index is maintained *incrementally*, the index is updated after every completed change to the table. Only the affected portions of the index are updated. Since the index is always up-to-date, queries and other location operations are substantially faster.

Incremental maintenance is especially attractive for large tables and those that are queried more often than they are changed. A complete list of events that cause secondary indexes to be incrementally updated is contained in Chapter 23 of the *User's Guide*.

Once a secondary index has been created, Paradox always tries to maintain it in the same way if possible. For example, an index maintained in batch mode will still be maintained that way even if you later specify maintained indexes in the CCP. Similarly, if you restructure a table, Paradox restructures its indexes in the same way. However, if you drop a table's key when you restructure it, its incremental indexes can no longer be incrementally maintained (because only keyed tables can have incrementally maintained indexes), so they are converted to non-maintained indexes.

Thus, the best way to change the way a secondary index is maintained is to use the PAL INDEX command to regenerate it and reverse the use of the MAINTAINED keyword.

No matter how an index is maintained, you can always use an INDEX command to regenerate it. Indexing takes a single pass through a table.

---

## Techniques

Understanding how *secondary* indexes work can give you a better feel for how they can best be used. Think of a secondary index as being similar to the index to a book—it gives you random access to a body of information according to a specified criterion or ordering.

Let's use a book on *Birds of the World* as an example. An index entry for "Birds" would be almost useless—it could conceivably contain citations for nearly every page in the book and therefore would not be very useful for readers trying to find information on a specific topic. You would have to sequentially search through all of the references cited in the index until you found exactly what you were looking for.

On the other hand, if the book contained only a single reference to the Kea (a carnivorous parrot found in New Zealand), and the index cited the exact page number on which the Kea was mentioned, you could locate that reference very quickly.

Secondary indexes behave in similar ways. If a particular field in a table contains many duplicate values, building a secondary index on that field may not provide much performance benefit in comparison to the cost of maintaining it. For example, suppose your *Customer* table contains 50,000 records. The number of discrete values in the State field can be no more than 50, and the average number of entries containing each value would be 1000. Under these circumstances, depending upon the type of processing you do with the table, it may not be very useful to create a secondary index on State. An index on the Zip Code field may prove to be much more powerful, since there are many more distinct values to index.



# Putting it all together

Now that you understand the building blocks and tools that are available, you're ready to put everything you've learned together and create Paradox applications. The three chapters in this part cover the following topics:

- Chapter 22, "Building Paradox applications." This chapter ties up the loose ends. You'll get a broad view of the steps you go through to design and create a finished application, and a discussion of how to package your application for the end user. The Paradox Runtime system is also described here.
- Chapter 23, "Multiuser applications." This chapter describes the implications of network use for PAL programs. It presents the complex issue of record locking in detail.
- Chapter 24, "The Sample application." This chapter walks you through the code used in typical sections of an application framework. The Sample application source code is provided on disk with Paradox.



# Building Paradox applications

It's time to put everything together. This chapter presents a brief summary of how you build a database application using Paradox and PAL. We'll explain the process, step-by-step, and note which portions of Paradox and PAL are applicable at each point. Finally, we'll describe what you need to provide your users so they can run your application.

Remember that the unique interaction of Paradox and PAL lets you do more than program complete applications. There are four levels at which you can use Paradox and PAL:

1. Create and use a database and its associated forms and reports interactively, using only the features of Paradox described in the *User's Guide*. The tables and other objects you create interactively can serve as the building blocks of your application.
2. Use Paradox's script recording and playback facilities to automate groups of interactive Paradox instructions. This process is described in Chapter 9.
3. Use PAL's SETKEY command to create simple keyboard macros for sequences of interactive Paradox operations, building shorthand keystrokes for common sequences. This process is described in detail in Chapter 20. You can then place the code for these SETKEY macros into scripts and procedures and store them in libraries.
4. Finally, use PAL commands and functions to enhance, extend, and customize programs into complete, standalone database applications. This is the process described in this chapter.

Although this book has focused on PAL as a sophisticated programming language, don't forget that it is also a tool to enhance your day-to-day interactive use of Paradox.

---

## How to create applications

Together, Paradox and PAL make up a complete development environment. You can create sophisticated, finished applications quickly by incorporating objects and code you generate interactively.

Here is one overall approach and strategy for developing applications using Paradox and PAL:

1. Using Paradox interactively, create the tables, queries, forms, and reports that will be the building blocks of your application. You can test these objects interactively and modify them until they meet your needs. If you are designing a multiuser application, begin implementing your data protection scheme now.
2. Determine the overall structure of the application by designing menus and dialogs that will guide the user through it. Use `SHOWPOPUP` and `SHOWPULLDOWN` to create the menus; use `SHOWDIALOG` to create dialog boxes.
3. For each operation in the structure you've designed, develop small, discrete components of the application using Paradox to record keystroke sequences and queries into scripts.
4. Use the PAL Script Editor or your own editor to write any components that can't be directly recorded by Paradox. Also, write any procedures you'll need that are unique to your application.
5. Link the components you've created so far by using PAL to tie things together and refine the menu system you designed earlier. Create procedure libraries that contain the bulk of your application, along with the scripts that control the use of those libraries.
6. Use the Script Editor to tie together the loose ends and link small sections of code that can't be connected easily using Paradox's recording features. You can write an errorhandling procedure and use the system variable `Errorproc` to link it to your application.
7. Throughout this process, use the Debugger to step through the application, making sure that sequences of steps are performed in the right order and that the values of variables and arrays are properly defined at each step.
8. Run the application a few times, noting any sequences where there are noticeable delays in execution. If this occurs in any section that was recorded using Paradox, check to see if there isn't a quicker way to do the same thing. If it occurs in performing a



query or LOCATE command, use INDEX to create a secondary index on the table.

9. If your application is a large one, encapsulate the top-level procedures and major modules into closed procedures.

Finally, you're ready to package the application for your users. If it is a multiuser application, assign auxiliary passwords to tables, classify users into groups, and develop password initialization scripts. If your application is to be distributed to others who do *not* own Paradox, apply for a Paradox Runtime license (described later in this chapter), create disks with your application's script and object files, as well as the Runtime files, and write documentation for your application.

Now let's look at each of the previously described steps in more detail.

---

## Design and create the building blocks

The very first step in creating an application is creating its foundation: tables, forms and reports. Decide on the tables you will need and how they should be structured. Since you can restructure Paradox tables whenever necessary, jump right in and begin creating tables for your application. After you've built a table and played with it interactively for a while, you can easily change the type of a field, add another field, or delete a field as needed.

Although you may want to key tables as you create them, don't worry about creating secondary indexes at this point. Paradox makes it easy to index tables after the fact. You need never worry about indexes during the prototyping phase of the development process. Later, when the table is structured the way you want it, you can add indexes to improve performance.

Once the tables are created and structured as you want them, use Paradox interactively to build forms for data entry, editing, and viewing, and to build reports for output. Use multi-table and multi-record forms so that you can have a single data entry form send data to multiple tables and consolidate data from many tables into one form for viewing and editing. Your forms and reports should reflect the needs of the user and not the underlying structure of the database.

Saved query statements are prime examples of the prototyping capabilities of Paradox. Once you've created the tables for your application, load them with a small amount of sample data and begin to develop queries. When they are just the way you want them, use Scripts | QuerySave to save the query statement. Later you can use the Script Editor to incorporate the saved query scripts into your application.

Remember that you can parameterize query statements by incorporating tilde variables either in Paradox or using the Script Editor. See “Query variables” in Chapter 19 for details.

If you are designing a multiuser application, you will want to begin implementing your data security system at this point. Later on you can assign owner and auxiliary passwords to the tables. Right now, however, note what classes of application users you’ll have (such as data entry clerks, data entry supervisors, application supervisors, and so on). You’ll want to structure the application so that data entry operators, for example, are given access only to data entry forms and not to reports. If you know who’s going to be using the application and what they’ll have access to, you’ll be far better prepared for creating the necessary forms and reports to match.

For further information about how data security affects your application, see

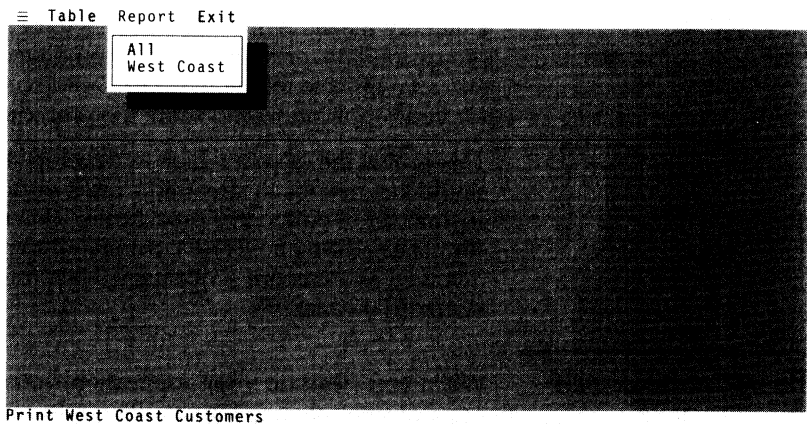
- “Password protection” in Chapter 7
- the discussion of data security in Chapter 21 of the *User’s Guide*
- the section on Tools | More | Protect in Chapter 17 of the *User’s Guide*

---

## Design the application’s structure

Once you’ve created the tables, queries, forms, and reports for your application, decide on its basic structure. In almost every case, you’ll want to use menus and dialogs to present the structure to the user. For example, a simple order entry application can be structured into three operations, in which the user is given a choice of working with system utilities, entering or editing data in tables, or preparing reports. Thus, the main menu could have the choices , Tables, and Report. If the user picks Tables, a submenu could offer a choice of table operations. If the user picks Report, a submenu might offer a choice of printing a report of all customers or only west coast customers:

Figure 22-1 Typical application menu



At this step, it's important to know what is on which menu, which menu leads to which, and which tables, forms, and reports are going to be used at each step.

---

### Record small components in scripts

Using Paradox, record short scripts that provide single modules of your overall design.

You can use either Scripts | Record or Instant Script Record *Alt-F3* to capture the sequence of commands for each module during the prototyping process. Then use Scripts | Play or Instant Script Play *Alt-F4* to play back the task and test it. Use the Debugger to step through the script if you suspect a problem.

Next, use the Script Editor (or your own text editor) to edit the resulting script to

- generalize recorded sequences; for instance, you can transform a recorded operation that modifies a single record to one that modifies an entire table by enclosing the code in a SCAN loop
- add comments to the recorded script, as appropriate
- replace any user-entered data, such as a table name, with variables that will be given a value in the initialization section of your application
- replace any user data entry or editing sections with WAIT commands
- add functionality that can't be captured through keystroke recording by using the PAL commands and functions

---

## Write additional procedures

Next, use the Script Editor or your own text editor to create those tasks that cannot be easily recorded with Paradox interactively. This generally means writing procedures to be performed on special groups of data, or to extract and combine data in special ways that are not available interactively in Paradox. Create a procedure library for your application and write the procedures you design into it.

This is also the step in which to develop shortcuts, such as creating single-key macros to streamline tasks or operations, or writing a custom set of data entry procedures. This might turn out to be as simple as adding a SETKEY command to a recorded task, or as involved as a custom script using elaborate GETCHAR() and FORMAT() functions.

---

## Tie up the loose ends

When you clean up your application, you might want to rename some of your choices for variables, add comments to some of the structures, and make sure that variables are used consistently. To do this, use either the Script Editor or your favorite ASCII text editor. Among other things, you'll want to make sure that

- ▮ Procedure libraries are up to date.
- ▮ No duplicate, misleading, or contradictory variable and array names are used.
- ▮ Variables are assigned, and arrays declared and assigned, before they are used.
- ▮ Comments clarify what is attempted at each step.

This is also the time to create a general error-handling procedure and link it into your application by assigning its name to the variable *Errorproc*. See "Error processing" and "Error procedures" in Chapter 7 for details on this process.

---

## Use the Debugger to test your work

The PAL Debugger, described in Chapter 10, is invaluable in getting your application to work flawlessly. At some point before you declare your application complete, you should run your application at least once with ECHO ON, and at least once in single steps using the Debugger. Pay careful attention to the values of variables and arrays at each point in the application (use Value to display them).

Make sure you test out the full range of potential user input, not just the correct responses. Try typing nonsensical data and responses to prompts. If your program doesn't handle inappropriate values and responses, you can

- ▮ add validity checks to your tables
- ▮ clarify instructions, prompts, messages, and help screens

- improve your error-handling procedure

---

## Tune the application

By this time you have a working database application. The question now becomes How well does it work, and can you make it better? Here are some things you can probably still improve at this point:

- Queries that take too long: Use the INDEX command to create secondary indexes on the affected tables.
- Users aren't sure what to enter into tables: Create entry forms for data, complete with full explanations of what is expected, or use ValCheck | AllCorrespondingFields | HelpAndFill to let users choose entries from a lookup table.
- Unclear or ineffective reports and forms: Use Paradox to redesign the forms and reports to make them more attractive and useful. You can redesign these objects at any time without affecting the underlying structure of the application.
- Large sections of recorded keystrokes: Replace these with abbreviated menu commands, where possible, to produce more readable scripts that are easier to change later.
- Tasks that execute slowly: Look for sections of code that perform a single, related task and have no dependencies on outside loops or control structures.
- Too many scripts in the final application: Look for PLAY commands in the application and use the Script Editor to convert scripts into procedures and store them in procedure libraries.

---

## Create closed procedures

If your application is large, finish the development process by making the top-level procedures and the major modules into closed procedures. Doing so helps your application take full advantage of Paradox's built-in memory management capabilities.

The method outlined here—or one like it which is adapted to your needs and work style—can help you to use PAL to build applications easily and efficiently. Use Paradox; it can do most of what you need to have done without the need to create special custom programs. Make it a practice to record your keystrokes while you conduct experiments on what Paradox can do. That way you'll have a reminder of what you tried when you get ready to finalize your application. Again, the more you use Paradox to prototype and model pieces of an application, the easier it will be to bring it to final form.

---

## Packaging your application

Once you've developed your application, you need to know how to package it for your intended user. To use your application, a user must have

- all necessary script files, pre-parsed scripts (.SC2 files), and procedure libraries
- all necessary Paradox objects that aren't created by the application itself
- a copy of Paradox or Paradox Runtime
- instructions on how to use the application
- (optional) a batch file to start the application
- (optional) a batch file to install the application

You can use DOS to copy all the required files onto a distribution disk.

If your application is designed to be run on a network by multiple users, remember that Paradox 2.0 or later must be supplied.

If you do not include pre-parsed scripts with your application, Paradox will create them the first time your application executes. However, your application will execute faster the first time if the pre-parsed scripts are included. If you do not distribute the .SC2 files and you install the application in a write-protected directory, Paradox must keep the application in memory.

---

### Script files

In order to make sure that all the necessary script files are included, use distinctive names for scripts in an application. For example, you might have each script name begin with an initial letter to indicate the application, a number to indicate the script number, and an underline. Then the remaining characters would give a brief description of what the script does. For example, for a personnel application, you might have script files named P0\_MAIN.SC, P1\_ADD.SC, P2\_EDIT.SC, P3\_RPRT.SC, and so on. Whatever method you use for naming, make sure you get all of the scripts that comprise your application. Use your editor and search for PLAY commands if you're not sure.

---

### Procedure libraries

Most applications consist primarily of procedure libraries controlled by one or more scripts. Like scripts, procedure libraries should have unique, identifiable names. If you are supplying your applications to users who are already running Paradox on their system, be careful of using names that might duplicate files they might already have. For

example, it's fine to store your procedures in a library called Paradox, but if users already have a library with the same name, copying your application to their system could cause their PARADOX.LIB file to be replaced.

---

## Paradox objects

Your distribution disk should include all necessary scripts, tables, libraries, reports, forms, validity checks, settings, and indexes that aren't created by the application itself. If you keep these in a separate directory as you're developing the application, it will be easy to find them when you're ready to package them. Of course, if you've loaded tables with sample data in order to test the application, make sure to empty them.

---

## Paradox or Paradox Runtime

Users who already own Paradox only have to install your application and run it. See the next section on "Initial batch file" for ways of making this easy. If your application uses PAL commands or Paradox features that have been added since the initial release of Paradox, remember that your user's version of Paradox must support the new commands or features.

For users who do not yet own Paradox, you have two choices:

- Require them to purchase a copy, which lets them work with data interactively, as well as through your application.
- Provide a copy of Paradox Runtime, which restricts them to running only completed applications.

The Runtime version of Paradox contains all the parts of the program necessary to run an application, but does not work interactively. Since it is only useful for running scripts, it is generally unseen by the user. This is a very low-cost way to provide your application to users who do not need a complete, interactive database system.

**Note** Do not make an illegal copy of your regular Paradox program files. Providing copies of your regular Paradox program files to your users is a violation of both international copyright laws and the Paradox license agreement, and is unethical. You can only distribute Paradox Runtime files.

You will find a Paradox Runtime Order Form and License Agreement in your Paradox package. If it's not there, or if you need more information, contact

- Borland International  
Customer Support  
1800 Green Hills Road  
P.O. Box 660001  
Scotts Valley, CA 95067-0001, USA

---

## Documentation

If you're distributing a large application, you should include complete user documentation. You should provide instructions on how to install and start the application, how to navigate the menus, and other information on special keypresses, data entry, error messages, and troubleshooting. You should also describe all table structures and all outputs.

---

## Initial batch file

If your application is meant for novice users, we recommend creating a batch file to run the application so that users don't need to know how to start Paradox. The batch file might be as simple as

```
CD \PDOX40\GIFT
PARADOX GIFT
CD \
CLS
```

Then, assuming that

- this batch file is called GIFT.BAT and is stored in the root directory
- Paradox is stored in the \PDOX40 directory, which is referenced by the DOS PATH command
- the user has copied your application files to the \PDOX40\GIFT directory
- the top-level script is named GIFT.SC

the user only has to type **gift** to run your application. The application should make sure that it sets the proper working directory; otherwise, it might not find all the files and data.



# Multuser applications

This chapter amplifies the discussion of the multuser features of Paradox described in Chapters 17 and 21 of the *User's Guide*, and introduces the PAL commands and functions that control them. It discusses

- general principles of network applications
- setting up the directories in which applications reside
- using passwords to support multiple levels of data security
- using explicit locks to control concurrent access to shared objects
- using multi-table forms in a multuser environment
- how to ensure data consistency

Before attempting to write a multuser application, it helps to have a basic understanding of Paradox's multuser capabilities. If you haven't already done so, you should read the following in the *User's Guide*:

- "CoEdit" in Chapter 11
- "Net" and "Protect" in Chapter 17
- Chapter 21

You should also review the section on "Password protection" in Chapter 7 of this book.

---

## General principles

Inherent in all multuser applications is a basic conflict between data integrity and the need to let many users access the same data simultaneously. You could ensure data integrity by allowing only one user at a time to use a table, but that would be highly inconvenient to others. At the opposite extreme, you could allow anyone on the

network to use any table at any time, but such a free-for-all would quickly lead to inconsistencies in the data. The goal in all multiuser applications is to ensure data integrity while maximizing concurrent access.

Paradox's automatic locking features are designed to strike the best possible balance between these goals in interactive use. They can go a long way toward providing the same balance in your applications. For example, if you use the WAIT command in Coedit mode, the automatic record locking features of this mode eliminate the need for your program to control locks explicitly. By taking advantage of Paradox's automatic locking features, it is possible to write a functional multiuser application without ever having to explicitly lock either records or objects. Nonetheless, you will usually want to explicitly lock shared resources so your application can take the appropriate actions if a lock fails.

---

### **Converting single-user to multiuser applications**

You can convert a single-user application to run in a multiuser environment by making thoughtful modifications to it. For example, in many instances, you will want to replace uses of Edit mode with Coedit to obtain the benefits of locking at the record level rather than the table level. Coedit mode provides record locking, which lets many people change the same table simultaneously, while Edit mode precludes more than one user from making changes at once.

Also, Coedit mode usually gives you an improvement in performance, even for single-user applications, because it does not maintain a complete transaction log. However, since Coedit mode operates differently than Edit mode, you can't always just substitute Coedit for Edit in your scripts.

In addition, you will usually need to add locking to your application. When you do so, be sure to test your application thoroughly; try to "break" locks. Bugs that involve locks show themselves only intermittently, when two network users happen to want the same resource at the same time. To thoroughly test a multiuser application, you'll need to duplicate network conditions.

---

### **Organizing a multiuser application**

When you set up a multiuser application, you must determine where the scripts, procedure libraries, tables, and other objects it includes will reside on the network. You also need to make sure that users have the appropriate operating system access rights to the directories and files involved in the application. The exact manner in which these rights are granted depends on the particular network on which your application will be running.

Your application should be laid out as follows:

- Shared tables and family objects should reside in a shared directory to which users have read/write/create access.
- The application's scripts and procedure libraries should reside in a shared directory to which users have at least read access. You should include all the .SC2 files if your application will reside in a read-only directory because Paradox will be unable to create them. See Chapter 21 for additional information about .SC2 files.

Users must have read/write/create privileges to the directories in which tables and other objects are stored to access these objects within the application. This is so because in order to lock an object Paradox creates a special lock file (with a .LCK extension) in the same directory in which the object is located.

Many network operating systems (and DOS itself) provide means of flagging individual files as read-only. The files containing Paradox tables and other objects may be flagged in this way to improve performance if they are in a locked directory. You might want to flag scripts and procedure libraries as read-only as well. If you do, you can store them in the same directory as the shared tables without having to worry about users accidentally modifying them.

---

## Private directories

In addition to shared objects, your application will generally need to use one or more private objects. As discussed in 21 of the *User's Guide*, each Paradox user on a network has a private directory. Paradox automatically stores the *Answer* table and other temporary tables it generates for a particular user in that user's private directory; otherwise, users would step on each other's toes.

The user's private directory should also be used to store scratch and dummy tables used by your application. You should not build any assumptions about the name of this directory into your application; you can expect that different users will have private directories with different path names. Instead, before you create the tables, use the PRIVTABLES command (described in the *PAL Reference*) to declare the tables private. Then, when you create the tables, Paradox automatically stores them in the user's private directory in exactly the same way as it stores Paradox temporary objects, although it does not delete them as it does temporary objects.

For example, suppose you use a table called *Rept* to store standard report specifications used in an application. You regularly empty this table and then use the ADD command to load the table with query results on which reports are based. Use PRIVTABLES to designate *MyRept* as a private table and copy *Rept* to it; this prevents contention problems when multiple users are running the application.

The PRIVDIR() function can be used to determine a user's current private directory, and the SETPRIVDIR command can be used to

change it. In most circumstances, however, you will not need these commands. By using PRIVTABLES to designate the names of private tables, you can refer to these tables without giving any drive or path, as if they were always stored in the current working directory.

---

## Controlling access to data

Since data security is not exclusively a multiuser issue, we've already discussed password protection in Chapter 7. Here we address multiuser aspects of data security.

Many multiuser applications require that different users have different degrees of access to data. For example, some users may be allowed to examine but not change data, while others can change existing records but not delete them or enter new ones, and still others are allowed to make whatever modifications they want.

Use Menu {Tools} {More} {Protect} {Password} or the PROTECT command to set up a nearly limitless variety of access levels for each table involved in an application. This command also encrypts the table, prohibiting users from accessing the table with an outside debugger or text editor.

To give users access to protected tables used in an application, use the ACCEPT command or an ACCEPT statement with the HIDDEN keyword in a SHOWDIALOG command to prompt the user for an application-level password. Then, behind the scenes, you can issue a PASSWORD command that provides the passwords for all of the protected tables.

You can also use Menu {Tools} {More} {Protect} to password-protect scripts. This only prevents users from viewing or modifying the script—not from playing it. The use of procedure libraries also affords protection, since libraries are stored in binary rather than text form.

For a complete discussion of password-protecting tables and other objects, see the discussion of Tools | More | Protect in Chapter 17 of the *User's Guide*. Also see the descriptions of IMAGERIGHTS, PASSWORD, PROTECT, UNPASSWORD, TABLERIGHTS(), FAMILYRIGHTS(), FIELDRIGHTS(), ISENCRYPTED(), and ISMASTER() in the *PAL Reference*.

---

## Managing locks

The key issue in designing any multiuser application is deciding how to resolve simultaneous requests for the same resources. Paradox's

locking capabilities give you the tools you need to manage concurrency without having to sacrifice either functionality or performance.

The guiding principle in planning your multiuser applications should be to minimize competition for resources. Here's how:

- Keep critical sections short; that is, don't tie up shared resources longer than you absolutely need to.
- Apply the weakest locks possible; for example, don't lock an entire table when locking a record suffices, and don't use a full lock when a write lock is sufficient.

In order to apply these principles, you need to understand exactly how locks work in Paradox.

Locks are used to limit access to a table or a record. Locks can be placed either automatically by Paradox when a particular operation is invoked, or explicitly by issuing one of a number of lock commands.

---

## Automatic locking

In a multiuser environment, Paradox automatically locks objects during every operation where there could be contention for a resource. In general, locks that are placed automatically by Paradox are the weakest possible consistent with the need to maintain data integrity for the duration of the operation.

For example, when the COPY command is invoked, Paradox places a write lock on the source table and on each of the objects in its family. The write lock prevents other users from modifying any member of the source family during the copy operation, but does not prevent them from accessing family members for read-only operations such as viewing. Paradox also places a full lock on the destination table involved in the copy; this lock prevents other users from accessing the destination table in any way during the copy.

In this instance, the destination table often does not even exist at the start of the COPY operation. Paradox can lock it nevertheless. This ability to lock a nonexistent resource is essential because it lets you prevent another user from creating an object during the period of time you are doing so yourself, or from deleting a table out from under you between the time you create it and the time you first use it.

---

## Explicit locking

Even though Paradox provides automatic locking for every operation, in most multiuser applications, you will want to use explicit locking commands to control access to resources in addition to depending on the automatic locks. The explicit lock commands give you more control than the automatic locks and also make it easier for you to handle situations where a user cannot place a lock because of contention for a resource with other users.

You can have explicit and automatic locks active at the same time on the same object. For example, if you use the LOCK command to explicitly place a full lock on the *Orders* table, and then use the COPY command to copy *Orders* to *Newords*, you will have placed both an explicit full lock and an automatic write lock on *Orders*. From the perspective of other users, *Orders* will appear to have only a full lock on it, since that is the stronger of the two locks you have placed. At the end of the COPY operation, the automatic write lock disappears, leaving your explicit full lock intact.

Other users can also place locks, both explicit and automatic, on objects that you have locked, so long as the locks they want to place can coexist with existing locks. Table 2-12 in Chapter 2 of the *User's Guide* shows which types of locks can coexist. Attempts to place object and record locks, both automatic and explicit, are honored on a first-come, first-served basis.

---

## Table locks

You can explicitly lock and unlock tables in two ways:

- by using Tools|Net|Lock and Tools|Net|PreventLock from the Paradox Main menu
- by using the PAL LOCK and UNLOCK commands (in programmed applications, the commands are more convenient to use)

You can explicitly place five kinds of locks on tables, each with a different strength. In order of increasing concurrency, they are

- A full lock prevents other users from accessing the table in any way—it gives you exclusive access. If you place a full lock on a table, no other user can choose that table from a Paradox menu. Consequently, they will be denied access to any objects in the table's family.
- A directory lock (DirLock) prevents users from modifying any Paradox object in that directory. It does not limit the ability to view tables or other objects in any way, but it prevents other locks and improves performance.
- A write lock prevents other users only from changing the contents of a table. It does not limit the ability to view the table, nor does it limit access to the objects in the family of the table in any way. A write lock also does not limit the ability of another user to place a write lock on the table, thus preventing you from writing.
- A prevent write lock prevents other users from placing a write lock or a full lock on a table, either automatically or explicitly. By securing a prevent write lock, you guarantee yourself the ability to access and make modifications to a table. (Other users can, however, lock individual records.)

- A prevent full lock prevents other users from placing a full lock on a table, either automatically or explicitly. By securing a prevent full lock, you guarantee yourself at least read-only access to the table.

The full lock and write lock limit the operations other users can perform on the table. They are appropriate when a table must remain stable for a given period. For example, if you intend to perform an operation that cannot tolerate changes to the contents of a table by other users, you should write lock it. If you intend to perform an operation that creates, deletes, sorts, or makes other massive changes to a table, you should probably place a full lock on it.

In addition to limiting access by other users, full locks and write locks can be used to improve performance. For example, if you write lock a shared table, many operations are faster because Paradox does not need to check whether other users have modified it. If you place a full lock on a shared table, operations are even faster because Paradox can buffer in memory the changes you make to a table, rather than having to write each one out as it is made (see the SAVETABLES command in the *PAL Reference* to find out how to clear the buffers). Thus, write locks and full locks are beneficial even if you do not expect others to access the table.

The directory lock also improves performance because it allows Paradox to treat any object within it as read only. Since no object in a locked directory can be modified, Paradox does not have to check objects for modifications. Paradox can then cache data for each user looking at tables in the locked directory.

A directory lock creates two lock files in the locked directory. The file PARADOX.LCK prevents Paradox 3.5 users from accessing the directory, and PDOXUSRS.LCK prevents Paradox 4.0 users from accessing the directory.

Unlike other locks, directory locks are not removed by Paradox when you exit; they must be explicitly removed. Do not delete the lock file at the DOS level while any version of Paradox is running on your network; use the menu command Tools | Net | Lock | DirLock, specify the directory, then clear the lock.

Prevent write locks and prevent full locks are useful for guaranteeing your own access to a table. For example, suppose you are about to use Coedit to modify several records in a table and you want to prevent other users from blocking any of your edits by, say, copying the table. (Remember that the COPY command automatically places a write lock on the table being copied; so, if you try to lock a record during the other user's COPY operation, your record lock fails.) By placing a prevent write lock on the table you are going to coedit, you are sure that other users cannot block your edits by write locking the

table. Of course, a prevent write lock would not prevent another user from locking a particular record that you want to edit.

Prevent full locks (the least restrictive type of lock) are similar to prevent write locks, but they preclude other users only from obtaining exclusive access to a table. They do not prevent other users from placing write locks. The VIEW command, for example, places an automatic prevent full lock on the table you are viewing. Placing a prevent full lock (or stronger lock) on tables that are rarely modified can help improve performance.

Used together, a write lock and a prevent write lock provide the optimal combination of locking and performance. This combination lets you modify the table and provides fast I/O; other users are prevented from modifying the table but are still permitted to view, query, and report from the table.

There are other table locks that are automatically placed by Paradox when you use linked tables in certain operations. See “Using multi-table forms on a network” later in this chapter for details.

---

### **Using LOCK and UNLOCK**

Use the LOCK and UNLOCK commands to place explicit table locks. Both LOCK and UNLOCK take as arguments a comma-separated list of one or more pairs of strings representing table names and keywords representing lock types. The following command, for example, places a full lock on the *Orders* table and a write lock on the *Stock* table:

```
LOCK "Orders" FL, "Stock" WL
```

The keywords WL and FL are used for write lock and full lock, respectively.

The LOCK command never does a halfway job. If you attempt to place more than one lock in a single LOCK statement, as above, it either places all the locks or none of them. This feature lets you avoid application deadlocks. As a side effect, LOCK sets the special system variable *Retval* to True or False, depending on whether all the locks were successfully secured.

In the event the locking attempt fails, the `ERRORCODE()`, `ERRORMESSAGE()`, and `ERRORUSER()` functions can be used to determine why. For example, if *Orders* were in use by another user named Ralph when you issued the preceding LOCK command, `ERRORCODE()` would return 3 (**Table in use**), `ERRORMESSAGE()` would return the string “**Orders table is in use by Ralph**”, and `ERRORUSER()` would return the string “**Ralph**”. `ERRORCODE()` always returns 0 if the lock succeeds.

You should always either interrogate *Retval* or use `ERRORCODE()` after attempting to place explicit locks unless you are absolutely sure



that the attempt will not fail. It is often convenient to place the LOCK command and the related test on *Retval* or ERRORCODE() in a WHILE loop, as

```
WHILE (True)
  LOCK "Employee"
  IF (Retval)                ; lock succeeded?
    THEN QUITLOOP           ; then continue beyond loop
  ELSE MESSAGE ERRORMESSAGE() ; show user the error message
ENDIF
...                          ; rest of module
ENDWHILE
```

What goes into the ELSE clause in this code depends upon the application. For example, it might be appropriate to ask users whether they are willing to wait for the table to be free. If you have structured the application in such a way that it will be tied up for only a very short period of time, you may just want to display a **Waiting...** message, pause for a second using the SLEEP command, then loop back to the top of the WHILE. You should avoid looping without first pausing, as this will load the network unnecessarily. You could also use the SETRETRYPERIOD command (described later in this chapter) to build an automatic retry period into the LOCK command itself.

As soon as an explicit lock is no longer needed, the UNLOCK command should be used to release it. For example, to undo the locks on the *Orders* and *Stock* tables in our earlier example, execute

```
UNLOCK "Orders" FL, "Stock" WL
```

You could also use

```
UNLOCK ALL
```

which releases *all* explicit table locks you had placed. The RESET command also releases all explicit table locks. Locks are not automatically released when script play ends (but are released when you exit from Paradox). It is recommended that you execute a RESET command near the end of your script in order to clear the workspace, undo all locks, and clear any passwords that have been presented during the course of running the application.

---

## **Deadlock**

There are situations in which contention for resources in a multiuser environment can lead to deadlock. For instance, suppose two users, Ken and Dick, both need to lock *Orders* and *Stock*. Say Ken has successfully locked *Orders* and is attempting to lock *Stock*; meanwhile, Dick has locked *Stock* and is attempting to lock *Orders*. Each of the two users is waiting on a resource held by the other, producing what is called a *deadlock*.

Because the LOCK command gives you the ability to lock more than one table at once, it has built-in deadlock prevention. If Ken executes

```
LOCK "Orders" FL, "Stock" FL
```

at exactly the same time that Dick executes

```
LOCK "Stock" FL, "Orders" FL
```

one of the two will secure a lock on both resources, while the other will fail to lock either.

The key to avoiding deadlock situations is to attempt to lock all the resources you need for a particular operation with a single LOCK statement. If you structure your applications by using small modules and by locking all the tables you'll need at once, you won't need to worry about deadlocks.

---

### **Multiple locks on one table**

You can explicitly place two or more different types of locks on the same table simultaneously. For example,

```
LOCK "Orders" WL, "Orders" PWL
```

attempts to place both a write lock and a prevent write lock on the *Orders* table. The write lock keeps other users from modifying the table; the prevent write lock keeps other users from blocking your modifications.

When applying multiple locks to a single table, make sure to pair each explicit lock with an explicit unlock of the same type. An object isn't unlocked until you remove as many locks as you placed. Note that placing two or more of the same type of lock on a single table does not alter the effect of the lock. However, the ability to apply multiple locks of the same type can sometimes simplify flow of control in programs. For example, it permits a script or procedure to perform a LOCK followed by an UNLOCK without undoing a lock already secured by a calling script or procedure.

---

### **Using LOCKSTATUS()**

The LOCKSTATUS() function can be used to determine whether (and how many times) you have explicitly placed a certain type of lock on a table. For example,

```
LOCKSTATUS("Orders", "WL")
```

returns 0 if you have not placed any write locks on *Orders*, or 2 if you have placed two write locks. The keyword ANY as the second argument lets you determine how many locks of whatever type you have placed on the table.

LOCKSTATUS() reports only your own locks—not those of other users. You can use Tools | Info | Lock from the Paradox Main menu to get a list of all the locks other users have placed on a table. However, the usefulness of this information may be of limited value in an application because another user or process can always place or release a lock an instant after your inquiry is made.

---

## Record locking

Coedit mode is unique in that it lets you lock individual records of a table. Locking a record prevents other users from modifying or deleting it for the duration of the lock; however, they can still view it. When another user moves to a record that you have locked, it has the same value as it did at the time the lock was placed. When you subsequently unlock the record, your changes are posted to the table and can then be seen by other users.

From the standpoint of an application, record locking serves two purposes:

- It provides you with the exclusive ability to make changes to a given record.
- It forces a refresh, so that you are guaranteed to be working with the most up-to-date field values.

It is important to recognize that Paradox's autorefresh mechanism is not active during script play except inside a WAIT command or within a GETEVENT loop. In addition, the REFRESH command (and the equivalent Refresh *Alt-R*) is not particularly useful in an application because another user might make a change an instant after the command is executed. Thus, even if the very next statement after the REFRESH interrogates the value of a field, you could still get an obsolete value.

Therefore, if you are in a situation in which other users might be changing or deleting a record you want to examine, lock it before examining it even if you don't intend to change it. In this way, you can guarantee that you are working with the record's current value. There are situations where your script is periodically examining a record waiting for a value to change. If so, you wouldn't want to lock everyone else out of the record, because then they wouldn't get a chance to change it. In this case, a REFRESH right before the test would be sufficient.

---

## How to lock records

Like table locks, record locks can be placed either automatically, by performing an action that modifies a record, or explicitly, by using the LOCKRECORD command. Similarly, record locks can be released either automatically by moving the cursor to a different record or explicitly by using UNLOCKRECORD.

The only situations in which you should rely on automatic record locking and unlocking within a programmed application are

- when you are sure the lock cannot fail (for example, if you are using Coedit mode for a single-user application or if you have placed a prevent write lock and a write lock on a table)

- inside a WAIT command, during which time the application has relinquished control to the user
- inside a GETEVENT/EXECEVENT cycle, during which time the application has relinquished control to the user

In all other circumstances, you should lock and unlock records explicitly, using the LOCKRECORD and UNLOCKRECORD commands. This is because, unlike Paradox's automatic record-locking mechanism, the LOCKRECORD command does not produce a script error if it cannot obtain a record lock because of a resource conflict. Instead, it sets *Retval* to False and sets an error code, which you can then interrogate with the ERRORCODE() function or the ERRORINFO command.

If you relied on automatic record locking in an application module that attempted to perform a field assignment (for example, [ ] = "Harry") in Coedit mode, a script error would be triggered if the automatic lock failed for any reason. While script errors can be handled by defining an error procedure to do so, it is much more straightforward to lock the record explicitly and then interrogate the error code.

In some cases, there are no automatic locks applied to satisfy the needs of an operation. For example, suppose your application lets users view, add, and delete records, but never change existing records. You might think that this code works:

```
MOVETO [Name]
LOCATE "John Doe"
IF (Retval)
  THEN DEL
ENDIF
```

This will almost always work—but isn't guaranteed to. If another user deletes the record the instant after it's found by LOCATE but before it's deleted by DEL, you'll get the message **Another user deleted record**. Here's a more thorough approach:

```
MOVETO [Name]
LOCATE "John Doe"
IF (Retval)
  THEN LOCKRECORD
  IF (Retval)
    THEN DEL
  ELSE MESSAGE "Can't lock record."
ENDIF
ENDIF
```

You can also lock and unlock records explicitly using the LOCKKEY command, which is the PAL equivalent of pressing Lock Toggle *Alt-L*. However, the LOCKRECORD and UNLOCKRECORD commands are both more versatile and less likely to result in coding errors, so we recommend strongly that you stick with them.

---

## Using LOCKRECORD and UNLOCKRECORD

In order to completely understand how LOCKRECORD and UNLOCKRECORD work, it helps to remember that there are two basic ways in which you can coedit a record:

- You can make changes to a record that already exists.
- You can enter a brand new record that has not yet been posted to the table.

If you are not familiar with coediting, you should read the “Coedit” section in Chapter 11 of the *User’s Guide*.

---

## Locking existing records

To examine or modify an existing record under script control in a multiuser environment, make the record current and then execute LOCKRECORD. You can then interrogate *Retval* to determine if the attempt at locking was successful. If the lock succeeded, *Retval* will be set to True; otherwise, it will be False. (You can also use the RECORDSTATUS() function to test if a record is locked—see the *PAL Reference*.)

In the event a lock fails, you can use the ERRORCODE() function to determine why. The most likely possibility is that some other user has already locked the record. Other possibilities are that another user has placed a write lock on the table or that the record you tried to lock was deleted. You can supplement the information returned by ERRORCODE() by using the ERRORMESSAGE() function, which returns an appropriate error message, such as **Record is locked by John**. You can also use the ERRORUSER() function to determine what user locked the record.

If the lock attempt is successful, the application can proceed to examine or change the record. When it is through processing the record, use POSTRECORD to post any changes and to specify whether or not the record should remain locked.

Like LOCKRECORD, UNLOCKRECORD sets *Retval* to True or False depending on whether it succeeded. Unless key fields of the locked record were modified to match the key of some other existing record (in which case, a key conflict would arise), the UNLOCKRECORD command will succeed.

---

## Using POSTRECORD

When you open up a new blank record by using *Ins* (or Down ↓ or *Ctrl-PgDn* at the end of a table) and begin entering data into it, the new record does not actually exist in the table until you post it. It’s important to remember that a record that has not yet been posted to a table cannot be locked. Indeed, if a record has not yet been posted, there is no need to lock it because other users can’t access it. When you are ready to post the record to the table, you can post it with POSTRECORD.

The full syntax of POSTRECORD is:

**POSTRECORD [ NOPOST | FORCEPOST | KEYVIOL ] [ LEAVELOCKED ]**

If there is no key violation, the record is always posted. If there is a key violation, the first parameter specifies what action to take:

- NOPOST does not post a record if a key violation occurs. If the first parameter is omitted, NOPOST is assumed.
- FORCEPOST always posts a record, even when a key violation occurs, unless the record is on a master table and a detail depends upon it. If the record has a dependent detail, a script error will occur.
- KEYVIOL leaves the record in a key violation state. This situation is described in the “Handling Key Conflicts” section below.

If the LEAVELOCKED parameter is specified, the record remains locked and current after the post. If LEAVELOCKED is not specified, the record is unlocked and Paradox may not keep it current (that is, the record may “fly away” to a new position in the index sequence). LEAVELOCKED is ignored if the post fails for any reason.

If a record is posted with POSTRECORD, *Retval* is set to True; otherwise, *Retval* is set to False.

POSTRECORD provides a high degree of control when coediting or posting records. A program can enter enough of a record to uniquely identify it on a network, post it to check for key violations, and then finish filling out the data.

You can also use LOCKRECORD and UNLOCKRECORD to post a record. In this context, the main difference between these two commands is that LOCKRECORD both posts and locks the new record, whereas UNLOCKRECORD just posts. In addition, in the case of a keyed table, LOCKRECORD makes the new locked record the current record in the table; this means that the cursor follows the record as it is posted in its key sequence. UNLOCKRECORD, on the other hand, only causes the new record to be posted; in the case of a keyed table, the record can “fly away” because the cursor moves to its rightful place in the key sequence of the table.

In either case, you can use *Retval* and ERRORCODE() to determine whether the operation succeeded. There are only two possible reasons for failure: either the table was write locked by another user or a key conflict occurred.

---

### **Handling key conflicts**

A key conflict arises when you do either of the following:

- try to post a new record with the same key as an existing one

- try to post a modification of an existing record that gives it the same key as another existing record

In either case, the posting operation fails and *Retval* is set to False. The `RECORDSTATUS()` function can also be used to test for a key violation—see the *PAL Reference* for details.

At this point Paradox automatically locks the existing record with the conflicting key. Thus, there are now two locked records you control—the new record you were just working on and the existing record. You can resolve the key conflict in one of three ways:

- Update the existing record by overwriting its non-key values with those of the new record.
- Change the key value of the new record so there is no longer a conflict.
- Delete the new record.

The technique of handling key conflicts is illustrated in Example 23-1. You can examine the values of the existing record by using the `KEYLOOKUP` command (equivalent to pressing *Lookup Alt K*). Doing so will alternately display the existing and new non-key values in the record. When you have settled on the version of the record you want to keep, reissue the posting command that signaled the key conflict in the first place. This time, the key conflict is ignored and the current contents of the row are posted. If you want, you can also use the `UNDO` command, which restores the record you are working on to its original condition and releases any record lock.

The automatic lock Paradox attempts to place on the existing record when there is a key conflict can fail. This will happen if another user has locked the conflicting record, or has placed a write lock on the table. You can distinguish among these cases by calling `ERRORCODE()` after the key violation has occurred.

#### Example 23-1 Locking records

---

Suppose you're writing an application that involves looking up a name in the *Employee* table. If a record already exists for the employee you are looking up, you want to lock it, perhaps to change some associated information in it, and then post the change. If no record yet exists, you want to add a new one.

If the user types an existing name, the `POSTRECORD` inside the `WHILE` loop fails with error code 53 (Key conflict). At this point, the existing record is automatically locked. Executing `KEYLOOKUP` now causes the existing record to be copied into the current row. Control then returns to the top of the `WHILE` loop, where re-execution of `POSTRECORD` closes the current row and moves the cursor to the existing record, which remains locked. After the user edits this row, a `POSTRECORD` command is issued, which posts the changes. The `POSTRECORD` is guaranteed to succeed, so there is no need to interrogate *Retval* again.

This approach is the recommended method for locating an existing record in a shared keyed table. The LOCATE command can also be used, but should be avoided if the record you are attempting to locate could be deleted or modified by another user. Whereas the technique shown in this example both locates *and* locks the record of interest in a single action, the LOCATE command only locates it.

In the instant between the execution of LOCATE and the command following it in your script, another user could delete or modify the record, which would in effect invalidate the LOCATE command. Moreover, LOCATE does not force a refresh, so that the located record could be out-of-date in any case.

See KEYLOOKUP, LOCKRECORD, and UNLOCKRECORD in the *PAL Reference* for other methods of handling key violations.

```
COEDIT "Employee"
END
DOWN                ; open up blank record at end
PICKFORM 2
WAIT RECORD         ; let user fill in name
  PROMPT "FILL in first and last names, then press F2"
UNTIL "F2"
WHILE True
  POSTRECORD KEYVIOL ; attempt to post and lock
  IF (Retval)
    THEN QUITLOOP    ; lock succeeded--name is new
  ENDIF              ; so leave WHILE loop

  ; find out why lock failed,
  ; was record locked (9) or was table write locked (3)?
  IF (ERRORCODE() = 9 OR ERRORCODE() = 3)
    THEN
      MESSAGE ERRORMESSAGE() + "... trying again"
    ELSE
      ; otherwise, there is key conflict (53)
      KEYLOOKUP      ; pick up existing record
      LOCKRECORD
    ENDIF            ; loop and lock rec again to confirm
  ENDWHILE           ; new or old record is locked
WAIT TABLE         ; let user fill in
MESSAGE "Edit info for this customer, press F2 when done"
UNTIL "F2"
POSTRECORD          ; cannot fail
DO_IT!              ; end the coedit
```

---

### **Multiple record locks**

In many applications, it is useful to lock records in different tables simultaneously. For example, if you are processing an order for a given customer, you may need to update inventory and billing tables as part of the same transaction. In order to keep these tables consistent with one another, you must lock the appropriate record in each table before making any changes to them.

When you move to a different image, a record lock placed in the image you are leaving remains intact. Therefore, no special handling is needed to simultaneously lock records in two or more tables. You only need to place each table on the workspace, then use the COEDITKEY command to invoke Coedit mode. Bear in mind that LOCKRECORD and UNLOCKRECORD apply to the current record



in the current image. When you leave Coedit mode, all record locks are released. Nevertheless, it is good practice to unlock each record explicitly because you must resolve any key conflicts before Paradox lets you out of Coedit mode.

You can also lock more than one record in the same table by displaying multiple images of that table on the workspace and locking a record in each image. (This works in Coedit mode on a network, but not in Coedit mode on a standalone machine.) In addition, as you will see in the next section, it is possible to lock an entire group of detail records in a restricted view by using a multi-table form.

---

## Using multi-table forms on a network

As described in Chapter 17, multi-table forms let you work with data from several related tables at once on a single form. These forms are very powerful because they are easy to construct and because they have built-in error checking to make sure that referential integrity is maintained. To facilitate using multi-table forms on a network, Paradox applies some special locking mechanisms to ensure that referential integrity is maintained even in a multiuser context. In order to take advantage of multi-table forms on a network, you need to be aware of these additional mechanisms.

---

## Simultaneous locking of multiple tables

In general, when you use a multi-table form, all of the tables included in the form are locked simultaneously. For instance, suppose that Form 3 of the *Customer* table is a multi-table form that incorporates embedded forms for *Orders* and *Products*. When you place *Customer* on the workspace to view it, a prevent full lock is placed on *Customer* as usual. However, as soon as you toggle to the multi-table form, *Orders* and *Products* will prevent full lock as well. This is the case even though *Orders* and *Products* were not explicitly placed on the workspace.

```
VIEW "Customer" ; prevent full lock placed on Customer
PICKFORM 3      ; displays form, locks Orders and Products
FORMKEY        ; toggles to table view, unlocks Orders and Products
```

Other than the fact that they imply simultaneous locking of multiple tables, for most operations, the locking rules applied to single tables apply equally to multi-table forms. This is, in fact, always the case with unlinked multi-table forms. The major exception is the use of linked multi-table forms in coediting, data entry, and form add operations.

---

## Group locks

Whenever a user or application is coediting through linked multi-table form and attempts to change the primary key of a master record, the group of detail records linked to that master record are locked. This kind of lock is called a group lock. Paradox places and

removes group locks automatically. The group lock essentially consists of a set of automatic record locks. The group lock places no significant additional restrictions on what other users can do with the tables that have been locked in this way.

The most important restriction is that for the duration of the group lock any other user wanting to coedit any of the group-locked records in the linked tables will be prevented from doing so. This rule ensures that the referential integrity checks established through the form will be placed on all users accessing the tables for update.

Other users or applications can coedit the master table and its linked detail tables through the same multi-table form, any other form, or even table view. If your application tries to begin a coedit operation on a group-locked record, error code 65 is returned when you are in table view and error code 9 is returned in form view.

**Note** Group lock and write record lock replace the form lock of Paradox 3.0 and 3.5.

---

### **Write record locks**

A write record lock is the complement of a group lock. Whenever a user or application is coediting a group of tables through a linked multi-table form and modifies a linked detail record or inserts a new linked detail record, the master record linked to those details is locked with a write record lock. This lock prevents modifications to the linked master record; it is placed and maintained automatically by Paradox.

Other users or applications can coedit both the master table and its linked detail tables through the same multi-table form, any other form, or table view.

---

### **Record locking in multi-table forms**

During a coediting session using linked multi-table forms on a network, several special rules are applied. These rules apply to links of all types (that is, one-to-one, one-to-many, many-to-one, and many-to-many).

- Whenever an attempt is made to lock a record in a linked detail table, an attempt is first made to lock the associated master record. If the master cannot be locked, the entire attempt fails.
- Conversely, whenever a user or session succeeds in locking a master record, no other user with the same form will be able to lock any of the associated detail records. In effect, all of the records in the restricted view in the detail are simultaneously locked.

The same rules apply whether Paradox is automatically trying to place a record lock or you explicitly try to lock a record in the context of a multi-table form using the LOCKRECORD command.

---

## The SETRETRYPERIOD command

As mentioned earlier, you should interrogate the value of *Retval* or `ERRORCODE()` after each attempt to place a lock unless you are sure the lock will succeed. In some cases, you might know from the flow of control in the application that a lock will eventually succeed if you try to place it repeatedly over a long enough interval of time. For example, in applications designed to run in a batch mode overnight, it may be acceptable to wait several minutes for a table to become available. In such cases, you can often simplify the flow of control by using `SETRETRYPERIOD`.

The effect of this command is to build an automatic retry period into every operation, implicit and explicit, that could fail because of a resource conflict. For instance, if you execute

```
SETRETRYPERIOD 100
```

then any operation you subsequently attempt that could fail because another user has a resource locked is automatically retried repeatedly for 100 seconds if a lock fails. For example, if the command

```
VIEW "Orders"
```

is executed during the time another user has placed a full lock on the *Orders* table, the `VIEW` will be retried until either the other user has released the lock, or 100 seconds have elapsed. If the lock can't be obtained before 100 seconds have elapsed, a script error results.

`SETRETRYPERIOD` should be used with caution. Your application must be prepared to handle the possibility that the retry period will elapse without success. Also, because the retry period is not reset when script play ends, be sure to disable the automatic retry feature when you no longer need it by executing

```
SETRETRYPERIOD 0
```

If your application leaves it set to a large number, a user who later attempts unsuccessfully to lock a table or record will be confused by the delay—indeed, Paradox will appear to have frozen. You can determine the length of the current retry period by using the `RETRYPERIOD()` function.

---

## Currency of data

When a user makes a change to a table, the change is reflected immediately on that user's screen. Changes made by other users are reflected

- when the user presses Refresh *Alt-R*, or when the script the user is playing executes a REFRESH command
- when Paradox executes an autorefresh
- when certain operations are performed, such as locking, deleting, or posting a change to a record in Coedit mode
- when the user enters a mode that requires exclusive access to a table, such as Edit mode
- when the user performs an operation that forces Paradox to go out to disk because the data required by the operation does not happen to be in memory at the time

Because the Paradox autorefresh feature is disabled during script play (except during a WAIT or a GETEVENT loop), the only way you can be certain that the data in a table is absolutely current is to write lock or full lock the table or record in question.

**Note** Simply executing the REFRESH command is not sufficient because it does not prevent other users from making changes to the table between the time you refresh and the time you interrogate the data in the image.

In many applications, you might know from the flow of control that a table or record cannot change in the middle of the operation you are performing. In some cases, you might not even care. Since locking and unlocking tables and records costs time, you can obtain greater efficiency in these situations by not locking.

---

## Query and report integrity

Suppose you perform a query, or print a report on a large table while others are still entering data into it. Are the results based on the original data in the table, or the new data that are still being entered? The answer depends on the current setting of the Tools | Net | Changes commands.

When you request a report with Tools | Net | Changes is set to Restart, Paradox takes a "snapshot" of the tables involved. Paradox places a write lock on the tables for the time that is necessary to complete this snapshot. Paradox does not actually begin processing your request until it has a complete snapshot of the data involved. If Paradox is unable to obtain a write lock, it will continue to retry up until the maximum number of times set by the SETRESTARTCOUNT command.

When you request a query with Tools | Net | Changes is set to Restart, Paradox places a write lock on each table for the time that is necessary. Paradox does not actually begin processing your request until it has a complete snapshot of the data involved. If Paradox is unable to obtain a write lock, it will continue to retry up until the

maximum number of times set by the SETRESTARTCOUNT command.

Like other mechanisms built into Paradox, these features guarantee referential integrity, so that the result of the operation is guaranteed to reflect a single, self-consistent state of the tables involved.

If Tools | Net | Changes is set to Continue, Paradox continues the query or report despite changes to the data involved. There may be errors or inconsistencies in the report or query, but the operation will finish. Inaccuracies in the results can be directly attributed to changing data. This option is not effective for outer joins or find queries. Delete, Insert, and ChangeTo queries are not affected by this option because they require full locks on the tables involved.

See Chapter 21 of the *User's Guide* for complete information about query and report integrity.

---

## Crosstab integrity

As in Tools | Net | Changes | Restart, Paradox places a write lock on the tables involved when you request a crosstab. A write lock is placed on the tables until the crosstab is complete. If Paradox is unable to obtain a write lock, it will continue to retry up until the maximum number of times set by the SETRESTARTCOUNT command.

See Chapter 21 of the *User's Guide* for complete information about crosstab integrity.

---

## Interprocess communication

In some multiuser environments, it is necessary for processes or applications to be able to determine what other processes or applications are currently doing. For example, there can be several processes that simultaneously feed updates into a single master table. The first feeder that starts up must initialize the entire environment for itself and for all other feeders. When each feeder starts, it must determine whether or not the initialization sequence has already been completed, and if not, whether another feeder is currently initializing.

While there is no PAL mechanism that supports explicit parameter passing between processes, by taking advantage of the fact that Paradox guarantees locking integrity of tables, there are two strategies you can use to achieve the same effect:

- Use the contents of a shared table to pass information between processes.
- Use the locks placed on a shared table as flags or semaphores to communicate between processes.

To employ the first strategy, set up a special table that will be used only internally. While the application is running, this table can be coedited by all the processes involved. You then can transfer information between processes in several different ways. For example,

- One process can write data into the table that other processes can then read.
- You can determine how many processes are currently active by having each write a record to the table when it starts up and delete the record when it completes. Other processes can determine the number by using the `NIMAGERECORDS()` function to see how many records are currently in the table.

The second strategy—using locks as semaphores—is useful in a multi-user environment when several users need to access the same information. For example, each user may have to increment an item number in a data entry application. If your application locks the record with the highest item number when it is accessed, that record can be incremented, posted, and unlocked by a single user and no-one else. Your lock acts as a semaphore to prevent other users from incrementing the item number at the same time.

---

## Using the SETBATCH command

The SETBATCH command is used to improve the performance of table updates in a multiuser environment. When update operations are performed after issuing a SETBATCH ON command, file I/O and concurrency control are minimized, resulting in improved performance.

**Warning** The SETBATCH command is intended to operate for less than two seconds. If another user attempts to update or access the current table after it has been changed, that user's system will freeze. The system will be frozen until a SETBATCH OFF is issued.

The SETBATCH command gives you exclusive access to a table for a short period of time. After you issue a SETBATCH ON for a table, no other user can access, open, modify, lock, or read from the table until you issue a SETBATCH OFF. SETBATCH is useful for several short operations that you want to occur sequentially. The SETBATCH command should be used by advanced developers for serializing operations and improving performance. Most developers will not need this command.

See the *PAL Reference* for examples and information about the SETBATCH command.

# The Sample application

The Sample application is a working, realistic example of the core components of a typical application. The scripts, tables, forms, and other files that you need to play the Sample application have all been created as part of the Paradox installation procedure and placed in the PDOX40\SAMPAPP directory.

This chapter contains the following:

- a description of how to use the Sample application
- an annotated code listing for both scripts in the Sample application

Read the section called “How to use the Sample application” in this chapter for instructions about using the Sample application as a data entry application. After you run the application to get a feel for how it works, you can examine the actual code—with our commentary—in the section called “Code listing and commentary” to see how the application was created!

---

## What is the Sample application?

The Sample application is a *fragment* of a typical data-entry application; it is not a fully-developed application. The Sample application is a working, realistic application, but its functionality is limited in many areas. We have intentionally limited this functionality to make it easy to demonstrate the core components of a typical application.

Although the functionality of the Sample application is limited, it is still a powerful application. We have simply skipped many of the “bells and whistles” that you would expect in a finely-tuned application. We have also simplified the event handling in the Sample application, to show you that the core of a working application can be constructed and used, while a more sophisticated event handler can be added to the application later.

---

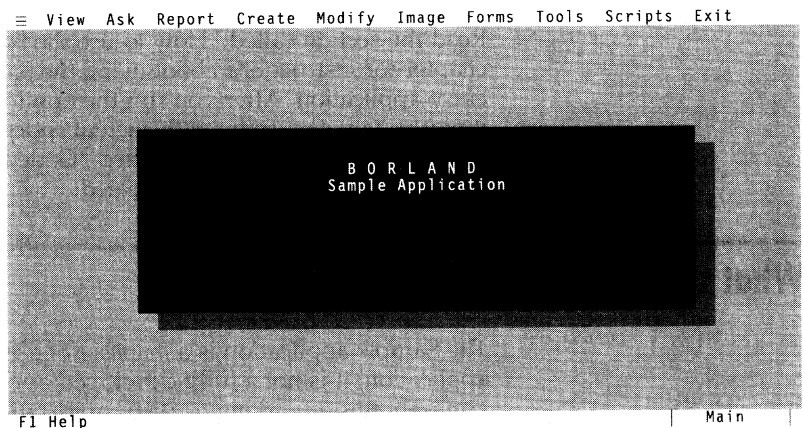
## How to use the Sample application

The Sample application is a fragment of a typical order-entry application. The main part of the application lets you enter customer orders and create invoices. Before you start to examine the code listing, run the Sample application and use it for awhile to see how it works. The following procedure describes how you would use the Sample application for data entry.

1. Specify your working directory by selecting Tools | More | Directory from the main menu bar of Paradox and entering `\PDOX40\SAMPAPP` in the dialog box that prompts you for a directory path.
2. Run the Sample application by selecting Scripts | Play from the main menu bar and entering **SAMPLE** in the dialog box that prompts you for the name of the script.

After a few moments, the screen clears and you see the splash screen shown in Figure 24-1.

Figure 24-1 Sample application splash screen



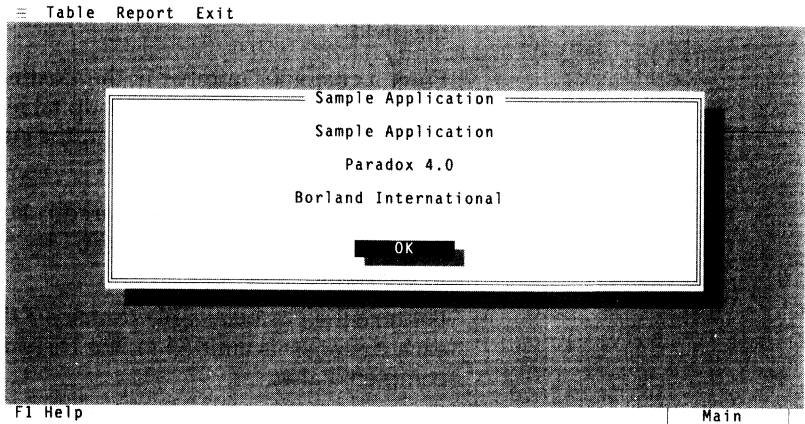
When the splash screen clears, you see the main menu bar of the Sample application.

3. Select `≡` | About from the main menu bar to display information about the Sample application.

The *about* box, shown in Figure 24-2, is displayed.



Figure 24-2 Sample application about box



When you opened the ≡ menu, you probably noticed the Utilities choice. This item is dimmed because it is not available in the Sample application.

Click OK to close the About box and display the main menu again.

4. Select Table | Modify from the main menu to begin creating customer orders.

The *Order Entry* form for the Sample application is displayed, as shown in Figure 24-3.

Figure 24-3 Sample application *Order Entry* form

5. Move to the Invoice No. field and enter an invoice number.

6. Move to the Order Date field and enter a date for the order.

Press *Space* or *Tab* to automatically enter the current date in this field.

7. Enter a customer number in the Customer No. field by pressing *F1* to display the *Customer* lookup table, placing the cursor on a customer, and pressing *F2* to put the customer number into the form.
8. Press Resync Key *Ctrl-L* and Paradox automatically supplies information for the Name, Addr, City, State, Zip, and Phone fields in the form.  
  
Because the *Customer* table contains a field listing the discount each customer is entitled to, the Discount % field is automatically completed also.

9. Move to the Payment Method field and complete it by pressing *F1* to display the *Payment* lookup table, placing the cursor on a method of payment, and pressing *F2* to put the payment method into the form.

If the method of payment is a credit card, move to the Credit Card Number and Exp. Date fields and complete them.

10. Move to the Ship Via field and complete it by pressing *F1* to display the *Carriers* lookup table, placing the cursor on a carrier, and pressing *F2* to put the carrier into the form.  
  
Because the *Carriers* table contains the cost of shipping for each carrier, the Shipping Charges field is automatically completed also.
11. Move to the No. field by pressing *F3* or using the mouse and enter a line item number for the first item.
12. Move to the Item field and complete it by pressing *F1* to display the *Stock* lookup table, placing the cursor on an item, and pressing *F2* to put the item into the form.  
  
Because the *Stock* table contains a description and price for each item, the Description and Each fields are automatically completed.

13. Enter the number of units the customer wants to order in the Quan. field.

When you leave the Quan. field, the value in the Amount field is automatically calculated and filled in.

14. Continue to enter line item numbers and quantities until the order is complete.

If you enter a duplicate line item number, the system displays an error message.

15. Move to the Shipping Instructions field by pressing *F3* or by using the mouse, press *Ctrl-F* or *Alt-F5* to enter field view, and enter any special shipping instructions in the memo field.

Press *F2* to save the shipping instructions and return to the *Order Entry* form.

16. Press Total *Ctrl-T* to calculate values for the Order Subtotal and Invoice Total fields and automatically complete them.

The Order Subtotal field displays the sum of the values in the Amount field. The Invoice Total field displays the total cost of the order after the discount amount has been subtracted and the sales tax and shipping charges have been added.

The Total key is available any time the cursor is positioned in one of the fields in the underlying *Invoice* table.

17. Move to the Sales Tax field and enter the sales tax percent.
18. Press Total *Ctrl-T* to recalculate a value for the Invoice Total field and automatically complete it.
19. Press *F2* to close the Order Entry form and return to the main menu of the Sample application, or press the *Ins* key to insert a new invoice and begin the order entry operation again.

---

## Code listing and commentary

The main part of the Sample application is a multi-table form that uses six tables:

- The *Customer* table contains the name, address, telephone number, and other related information about a large group of regular customers of the company that uses this application. The *Customer* table is used as a lookup table that lets the user quickly supply the necessary customer information for the main form. To simplify the Sample application, this table is read-only; you cannot add new customers to it.
- The *Stock* table contains the inventory information of the company that uses this application, including the item number, description, quantity, and price.
- The *Carriers* table contains seven possible methods of shipping the order, and the associated cost for each method.
- The *Payment* table contains seven possible methods of payment for the order.

- ❑ The *Invoice* table is empty when the application begins. When a user runs the application and creates customer orders, data is entered into the *Invoice* table.
- ❑ The *Orders* table table is empty when the application begins. When a user runs the application and creates customer orders, data is entered into the *Orders* table.

The Sample application contains two scripts:

- ❑ `sample.sc`, the main script
- ❑ `rptall.sc`, an instant script that was recorded and included in the application

The code listing for both scripts in the Sample application is provided in the sections that follow, with annotations explaining what happens in each part of the scripts.

---

## Code listing for SAMPLE.SC

```

1 ; Script      : SAMPLE.SC
2 ; Description : Paradox sample application
3
4
5 ; Begin by clearing any stray procedures and/or variables which
6 ; may have been created in previous Paradox work.
7 RELEASE VARS ALL
8 RELEASE PROCS ALL
9 ; Empty the desktop to make sure the workspace is in Main mode, revoke
10 ; passwords in effect but cancelled, flush memory buffers, write changes
11 ; to disk, and remove explicit table locks.
12 ALTSPACE {Desktop} {Empty}
13 ECHO NORMAL      ; display the empty desktop
14 ECHO OFF         ; then turn echo off to freeze the screen display
15
16 ; -----
17 ; Procedure   : Main()
18 ; Description :
19 ;   When a user plays the SAMPLE.SC script, the Main() procedure is called
20 ;   to start the application. The Main() procedure displays a simple
21 ;   "splash" screen describing the application, then displays the
22 ;   application's main menu. The user interacts with the main menu, and
23 ;   the Main() procedure performs appropriate actions based on the item
24 ;   selected from the menu.
25 ; -----
26
27 PROC Main()
28   PRIVATE MenuItemSelected,
29           ExitApplication
30
31   ; Draw a "splash" screen.
32   ShowSplashScreen()
33
34   ; Display and maintain the application's main menu
35   DisplayMainMenu()
36
37   ; Now that we've displayed the menu, we wait until the user makes a
38   ; selection from the menu and process it accordingly. We repeat

```

```

39 ; this process, executing the commands contained within the
40 ; WHILE...ENDWHILE loop, until the user wants to exit the application.
41 ; We use the logical variable ExitApplication as a signal to tell us
42 ; when to exit the loop. Initially we assign the value False to
43 ; ExitApplication so we'll enter the loop the first time through.
44 ; Each time we return to the start of the WHILE loop to process
45 ; another menu selection, we check the value of ExitApplication. If
46 ; the value is still False we enter the loop once again, processing
47 ; another menu selection. Otherwise, we exit the loop, executing
48 ; the first statement following the ENDWHILE at the end of the loop.
49 ; When the user selects Exit|Yes from the main menu, we assign the
50 ; value True to the variable ExitApplication.
51 ExitApplication = False
52
53 WHILE (ExitApplication <> True)
54
55 ; Grey out items which are not applicable from the main menu.
56 GreyMainMenuItems()
57
58 ; Wait for the user to select an item from the main menu. The menu
59 ; was displayed by DisplayMainMenu().
60
61 GETMENSELECTION TO MenuItemSelected
62
63 ; Examine the MenuItemSelected and execute the proper procedure or
64 ; (in the case of Exit Selections) set the proper flag. Note that
65 ; the string in the switch statement must be typed in EXACTLY as
66 ; it is in the menu procedure itself, including case and spaces.
67 SWITCH
68 CASE (MenuItemSelected = "Exit/Yes") : ; user wants to exit
69 ExitApplication = True
70
71 CASE (MenuItemSelected = "Exit/No") : ; user has decided not to exit
72 ExitApplication = False
73
74 CASE (MenuItemSelected = "AltSpace/About") : ; user wants info about
75 ; program
76 DisplayAboutBox()
77
78 CASE (MenuItemSelected = "Table/Modify") : ; user wants to modify
79 LetUserModifyTable() ; the Invoice table
80
81 CASE (MenuItemSelected = "Report/All") : ; user wants to create
82 PrintAllRecords() ; a report on ALL
83 ; customers
84
85 CASE (MenuItemSelected = "Report/West Coast") : ; user wants to
86 PrintWestCoastRecords() ; create a report
87 ; on West Coast
88 ; customers only.
89
90 ENDSWITCH
91
92 ENDWHILE
93
94 ; Before we leave the program we explicitly remove the menu. CLEARPULLDOWN
95 ; is not required here, but using the command is good programming
96 ; practice.
97 CLEARPULLDOWN
98
99 ENDPROC
100
101
102
103 ; -----
104 ; Procedure : DisplayMainMenu()

```

```

105 ; Description :
106 ;   Displays the main menu for the application. Note that the tags created
107 ;   in this procedure must be typed EXACTLY in the calling procedure for
108 ;   them to be processed properly.
109 ; Called By   : Main()
110 ; -----
111
112 PROC DisplayMainMenu()
113
114     ; Display the main pulldown menu.
115     SHOWPULLDOWN
116     "≡" : "Program Information." : "AltSpace"
117         SUBMENU
118             "Utilities" : "System Utilities" : "Utilities"
119             SUBMENU
120                 "Calculator" : "Popup Calculator" : "Calculator".
121                 "Help Interval" : "Set AutoHelp Interval" : "Help Interval"
122             ENDSUBMENU,
123             SEPARATOR,
124             "About" : "About this application." : "AltSpace/About"
125         ENDSUBMENU,
126         "Table" : "Work with the Invoice table." : "Table"
127         SUBMENU
128             "Modify" : "Modify Invoice table." : "Table/Modify".
129             "Close" : "Close Invoice table." : "Table/Close"
130         ENDSUBMENU,
131         "Report" : "Report on Customer Table." : "Report"
132         SUBMENU
133             "All" : "Print report of all customers." :
134                 "Report/All".
135             "West Coast" : "Print report of only West Coast customers" :
136                 "Report/West Coast"
137         ENDSUBMENU,
138         "Exit" : "Exit this application." : "Exit"
139         SUBMENU
140             "No" : "Do not exit application." : "Exit/No",
141             "Yes" : "Exit this application." : "Exit/Yes"
142         ENDSUBMENU
143     ENDMENU
144
145 ENDPROC
146
147
148
149 ; -----
150 ; Procedure : GreyMainMenuItems()
151 ; Description : This procedure greys out (dims) menu items which are not
152 ; applicable from the Main Menu. These items are no longer
153 ; selectable from the menu.
154 ; Called By   : Main()
155 ; -----
156
157 PROC GreyMainMenuItems()
158
159 ; Disable items not available in the Sample application. The Utilities
160 ; item is not available in the Sample application. The Table/Close
161 ; item is available only when a table is on the workspace.
162 MENUISABLE "Utilities"
163 MENUISABLE "Table/Close"
164
165 ; Enable main menu selections
166 MENUENABLE "Table/Modify"
167 MENUENABLE "Report/All"
168 MENUENABLE "Report/West Coast"
169 MENUENABLE "Exit/No"
170 MENUENABLE "Exit/Yes"

```

```

171
172 ENDPROC
173
174
175
176 ; -----
177 ; Procedure : DisplayAboutBox()
178 ; Description :
179 ;   Creates a simple dialog box displaying information about the
180 ;   application. It uses the SHOWDIALOG command to create the box
181 ;   with a single button which, when pressed, clears the box.
182 ; Called by : Main(),
183 ; -----
184
185 PROC DisplayAboutBox()
186
187 ; Create a dialog box with the following dimensions...
188 SHOWDIALOG "Sample Application"
189 @ 5, 10
190 HEIGHT 12
191 WIDTH 60
192
193 @ 1, 0 ?? FORMAT("W58,AC", "Sample Application")
194 @ 3, 0 ?? FORMAT("W58,AC", "Paradox 4.0")
195 @ 5, 0 ?? FORMAT("W58,AC", "Borland International")
196
197
198 ; The following creates the single OK button at the noted
199 ; coordinates. The "~" character assigns Alt-0 as the button hot key.
200 ; Because this is the only push button in the dialog box there
201 ; is no need to analyze the results.
202 PUSHBUTTON @ 8, 23
203 WIDTH 12
204 "~O~K"
205 OK
206 VALUE "OKButtonPressed"
207 TAG "OKButton"
208 TO OKButton
209
210 ENDDIALOG
211
212 ENDP
213
214
215
216 ; -----
217 ; Procedure : LetUserModifyTable()
218 ; Description:
219 ;   Wait on table. Allow the user to work with the
220 ;   table until s/he presses Formkey, Do It!, Del,
221 ;   Ctrl-T, or F10. F10 activates the SHOWPULLDOWN menu and
222 ;   temporarily suspends the WAIT until the menu interaction
223 ;   is complete.
224 ; Called By : Main
225 ; -----
226
227 PROC LetUserModifyTable()
228
229 ; Place the Invoice table in CoEdit mode, pick the "F" form, and
230 ; set the window conditions.
231 COEDIT "Invoice"
232
233 ; Move the table to a location where the user will not accidentally
234 ; display it.
235 WINDOW MOVE GETWINDOW() TO -100, -100
236

```

```

237 PICKFORM "F"
238
239 ; Get the handle of the form window.
240 WINDOW HANDLE CURRENT TO InvoiceWindow
241
242 ; Specify attributes of the form window in a dynamic array.
243 DYNARRAY WindowAttributes[]
244 WindowAttributes["HEIGHT"] = 21
245 WindowAttributes["ORIGINROW"] = 1
246 WindowAttributes["ORIGINCOL"] = 0
247 WindowAttributes["CANMOVE"] = True
248 WindowAttributes["CANRESIZE"] = True
249 WindowAttributes["CANCLOSE"] = False
250
251 ; Now apply these attributes to the form window
252 WINDOW SETATTRIBUTES InvoiceWindow FROM WindowAttributes
253
254 ; Enable/Disable menu selections.
255 GreyModifyMenuItems()
256
257 ; Wait on the workspace table. The prompt line is placed at the
258 ; bottom of the screen, telling the users what shortcut keys are
259 ; available.
260 ; When an event is "trapped" by the wait, the procedure
261 ; "ProcessWaitEvents" is called. This procedure acts as a handler,
262 ; deciding what to do with the particular event. In turn
263 ; a value of 0, 1, or 2 is returned to the wait.
264 ; 0 - processes the event and proceeds to the next event in the
265 ; trigger cycle.
266 ; 1 - does NOT process the current event and breaks out of the
267 ; trigger cycle, but does NOT break out of the wait.
268 ; 2 - does NOT process the current event and breaks out of the
269 ; trigger cycle AND the wait.
270
271 PROMPT " [F2] Close | [F10] Menu | [Ctrl-T] Update Totals |" +
272 " [Ctrl-L] Update Customer" ; set PROMPT
273 MESSAGE "Press [F1] for lookup help in fields followed by *" ; and MESSAGE
274 WAIT WORKSPACE
275 PROC "ProcessWaitEvents" ; call the wait procedure when any of the
276 ; specified events occurs. The procedure must
277 ; return a 2 to break out of the wait.
278
279 ; Trap the following keypresses. Note that we use numeric keycodes
280 ; that are explained in the comment following the KEY statement.
281 KEY -83, -60, 15, 20, -24, -18, -31, 26, -44, -97
282 ; Del, Do_It!, Ctrl-0, Ctrl-T, Alt-0, Alt-E, Alt-S, Ctrl-Z, Alt-Z, Ctrl-F4
283
284 ; Trap the following message event. If the user interacts with a
285 ; SHOWPULLDOWN menu and either selects an item or presses a MenuKey,
286 ; a MENUSELECT message will occur and the WAIT procedure will be
287 ; called.
288 MESSAGE "MENUSELECT"
289 ENDWAIT
290
291 PROMPT "" ; clear the PROMPT after leaving the WAIT
292 DO_IT! ; close the edit session
293 CLEARALL ; clear the table from the workspace.
294
295 ; Because this is a sample application, we empty the Invoice and Orders
296 ; tables so they can be used again.
297 EMPTY "INVOICE"
298 EMPTY "ORDERS"
299
300 ; ECHO NORMAL displays the empty workspace. ECHO OFF hides the
301 ; workspace again but leaves the empty workspace displayed on the full
302 ; screen canvas.

```



```

303     ECHO NORMAL
304     ECHO OFF
305
306 ENDPROC
307
308
309 ; -----
310 ; Procedure   : GreyModifyMenuItems()
311 ; Description : This procedure greys out (dims) items on the SHOWPULLDOWN
312 ;             menu which are not available while the user is coediting
313 ;             the multi-table form. Greyed items cannot be selected
314 ;             from the menu.
315 ; Called By   : LetUserModifyTable()
316 ; -----
317
318 PROC GreyModifyMenuItems()
319
320     ; Disable the following menu items.
321     MENUDISABLE "Table/Modify"
322     MENUDISABLE "Report/All"
323     MENUDISABLE "Report/West Coast"
324     MENUDISABLE "Exit/No"
325     MENUDISABLE "Exit/Yes"
326
327     ; Enable the following menu item.
328     MENUENABLE "Table/Close"
329
330 ENDPROC
331
332
333
334 ; -----
335 ; Procedure   : ProcessWaitEvents()
336 ; Description :
337 ;             Proc is called when an event is triggered from the modify table wait.
338 ;             The procedure evaluates the kind of message and does the appropriate
339 ;             processing, then returns a value of 0, 1, or 2 to the wait (See
340 ;             LetUserModifyTable proc for details on the return values).
341
342 ; Called By   : LetUserModifyTable()
343 ; -----
344
345 PROC ProcessWaitEvents(TriggerType, EventInfo, CycleNumber)
346
347 ; The TriggerType argument is assigned a value based on the type of event
348 ; or trigger that the procedure was called for. The dynamic array
349 ; EventInfo contains information about the event which occurred during
350 ; the wait. If the EventInfo element "Type" is "KEY", then we examine the
351 ; element "KEYCODE" to determine the numeric keycode that the WAIT command
352 ; trapped. For example, if the user pressed Ctrl-Z, the dynamic array will
353 ; contain the following elements:
354 ;     Dynarray EventInfo[]
355 ;     EventInfo["TYPE"] = "KEY"
356 ;     EventInfo["KEYCODE"] = 26
357
358 SWITCH
359     CASE (EventInfo["TYPE"] = "KEY") :
360
361         ; The following SWITCH...CASE...ENDSWITCH statement performs a
362         ; series of tests to determine what key was pressed and what mode
363         ; Paradox was in when the key was pressed.
364         SWITCH
365
366             ; The following CASE executes when the user presses the
367             ; Delete key (Keycode -83). This CASE calls the custom "Confirm
368             ; Delete" dialog box before deleting the invoice.

```

```

369 ; Because the delete is processed in the DisplayDeleteBox()
370 ; procedure, we return 1 to go back to the WAIT, deny the
371 ; pending event, and cancel the current trigger cycle.
372 CASE (EventInfo["KEYCODE"] = -83) : ; Del
373     DisplayDeleteBox()
374     RETURN 1
375
376 ; The following CASE executes when the user presses the
377 ; Do_It! F2 key. Each IF statement within this CASE tests
378 ; to see what state Paradox was in when the F2 key was
379 ; pressed, so the application can take the appropriate action.
380 CASE (EventInfo["KEYCODE"] = -60) : ; Do It!
381
382     ; It is possible that the user was in Field View, and
383     ; chose F2 to get out of Field View, not out of the
384     ; current wait.
385     IF (ISFIELDVIEW() = True)
386         THEN RETURN 0
387     ENDIF
388
389     ; It's also possible the user was in LookupHelp, and
390     ; pressed F2 to select a value from the lookup table.
391     IF (HELPMODE() = "LookupHelp")
392         THEN RETURN 0
393     ENDIF
394
395     ; It's possible the user was in CoEdit mode. We can't
396     ; leave CoEdit if the field data is invalid (e.g.
397     ; invalid date or incomplete picture). Before we
398     ; leave, we want to make sure that the value in
399     ; the current field is valid.
400     IF (ISVALID() <> True)
401         THEN RETURN 0
402     ENDIF
403
404     ; If the record passes the previous validity test,
405     ; we execute a Do It! to try to post the record.
406     ; The following IF statement then evaluates the mode
407     ; the system is in. If the system is in Main mode, the
408     ; Do_It! was successful and we terminate the WAIT
409     ; with a RETURN 2.
410     Do It!
411     IF (SYSMODE() = "Main")
412         THEN RETURN 2 ; exit the WAIT
413     ENDIF
414
415     ; If the Do_It! does not put the system in Main mode
416     ; and terminate the WAIT, a key violation exists. We then
417     ; issue an ECHO NORMAL to display the desktop to the user,
418     ; issue a Do_It! again so Paradox displays the KeyViol
419     ; message to the user, and RETURN 1 to go back to the
420     ; WAIT to let the user resolve the KeyViol.
421     ECHO NORMAL
422     Do It!
423     RETURN 1 ; couldn't leave CoEdit, let user resolve KeyViol
424
425 ; The following CASE executes when the user presses the
426 ; Ctrl-T Total key (Keycode 20). This CASE calls the proc
427 ; that totals the order lines.
428 CASE (EventInfo["KEYCODE"] = 20) : ; Ctrl-T (Total)
429     CalculateOrderTotal()
430     RETURN 1
431
432 ; The following CASE executes when the user presses
433 ; Ctrl-0 or Alt-0. This CASE denies the event so the user
434 ; doesn't inadvertently shell to DOS.

```

```

435         CASE (EventInfo["KEYCODE"] = 15) OR      ; Dos
436             (EventInfo["KEYCODE"] = -24) :      ; DosBig
437             ; Stay within the wait but don't process any more
438             ; events in the current cycle.
439             RETURN 1
440
441         OTHERWISE:
442             SOUND 400 100 ; issue a beep in other situations
443             RETURN 1
444
445     ENDSWITCH
446
447     ; If the EventInfo["MESSAGE"] element is "MENSELECT", then the user
448     ; has chosen from the menu.
449     CASE (EventInfo["MESSAGE"] = "MENSELECT") :
450
451         ; The following SWITCH...CASE...ENDSWITCH statement performs a
452         ; series of tests to determine what the user chose from the menu.
453         SWITCH
454
455             ; The following CASE executes when the user chooses =|About.
456             ; This CASE calls a proc to display a dialog box containing
457             ; information about the application. It then returns 0 to
458             ; go back to the WAIT.
459             CASE (EventInfo["MENUTAG"] = "AltSpace/About") :
460                 DisplayAboutBox()
461                 RETURN 0
462
463             ; The following CASE executes when the user chooses
464             ; Table|Close. The IF statement tests to see if the user is
465             ; in Field View (in the Shipping instructions memo) and closes
466             ; the memo, if necessary.
467             CASE (EventInfo["MENUTAG"] = "Table/Close") :
468                 IF (ISFIELDVIEW() = True) = True) ; if the user is in Field View
469                     THEN Do It! ; save the memo and close it
470                 ENDIF
471
472             ; We execute a Do It! to try to post the record.
473             ; The following IF statement then evaluates the mode
474             ; the system is in. If the system is in Main mode, the
475             ; Do It! was successful and we terminate the WAIT
476             ; with a RETURN 2.
477             Do It!
478             IF (SYSMODE() = "Main")
479                 THEN RETURN 2 ; exit the WAIT
480             ENDIF
481
482             ; If the Do It! does not put the system in Main mode
483             ; and terminate the WAIT, a key violation exists. We then
484             ; issue an ECHO NORMAL to display the desktop to the user,
485             ; issue a Do It! again so Paradox displays the KeyViol
486             ; message to the user, and RETURN 1 to go back to the
487             ; WAIT to let the user resolve the KeyViol.
488             ECHO NORMAL
489             Do It!
490             RETURN 1 ; couldn't leave CoEdit, let user resolve problem
491
492         OTHERWISE:
493             SOUND 400 100
494             RETURN 1
495
496     ENDSWITCH
497
498     ; This is the proc's safety valve. If the wait sends a message
499     ; we haven't planned on, then beep, process any further events
500     ; in the cycle, and remain within the wait.

```

```

501         OTHERWISE :
502             SOUND 400 100
503             RETURN 1 ; for safety, ignore events we don't recognize
504
505     ENDSWITCH
506
507 ENDPROC
508
509
510
511 ; -----
512 ; Procedure : DisplayDeleteBox()
513 ; Description :
514 ;     Brings up a dialog box prompting the user to confirm that s/he
515 ;     wants to Delete the record. If the user chooses Ok, then we
516 ;     can go ahead and delete the user. If not, then just return.
517 ; Called By : ProcessWaitEvents()
518 ; -----
519
520 PROC DisplayDeleteBox()
521
522     ; Define the "Delete" dialog box.
523     SHOWDIALOG "Delete Record"
524         @ 6, 10
525         HEIGHT 7
526         WIDTH 54
527
528     ; Text within the box itself.
529     @ 1, 4 ?? "Are you sure you want to delete this record?"
530
531     PUSHBUTTON @ 3, 30 WIDTH 12
532         "~C~ancel"
533         CANCEL
534         VALUE "CancelButton"
535         TAG "Cancel"
536     TO CancelButton
537
538     ; Place the OK and CANCEL buttons.
539     PUSHBUTTON @ 3, 11 WIDTH 12
540         "~O~K"
541         OK
542         VALUE "OK"
543         TAG "OKButton"
544     TO OKButton
545
546     ENDDIALOG
547
548     ; If the user accepted the dialog, then we go ahead and delete
549     ; this Invoice record. We then issue an ECHO NORMAL so the user
550     ; can see Paradox display the Delete message.
551     IF (RetVal = True)
552         THEN ECHO NORMAL
553             Del
554     ENDIF
555
556 ENDPROC
557
558
559
560 ; -----
561 ; Procedure : CalculateOrderTotal()
562 ; Description:
563 ;     When the user presses Ctrl-T, the line items are totaled and added
564 ;     to the relevent order fields to create an Invoice total.
565 ; Called By : ProcessWaitEvents()
566 ; -----

```

```

567
568 PROC CalculateOrderTotal()
569 PRIVATE LineTotal
570
571 ; initialize the line item amount
572 LineTotal = 0
573
574 ; If we're not on the Invoice table, then inform the user that Order
575 ; totalling can only occur from there.
576 IF (TABLE() <> "Invoice")
577 THEN BEEP
578 MESSAGE "You must be in the Invoice table to calculate totals"
579 SLEEP 2500 ; pause to display message
580 MESSAGE "" ; clear the message
581 RETURN
582 ENDIF
583
584 ; we can subtotal the line items. Move to the Orders table.
585 MOVETO "Orders"
586
587 ; calculate the extended price for each item number ordered,
588 ; and accumulate the running line total in the LineTotal variable.
589 SCAN
590 IF ((ISBLANK([Quantity]) = False) AND ; only update totals for
591 (ISBLANK([Item No.] = False)) ; records with qty and item
592 THEN [Amount] = ROUND([Unit Price] * [Quantity], 2)
593 LineTotal = ROUND(LineTotal + [Amount], 2)
594 ENDIF
595 ENDSCAN
596
597 ; Now we need to update the Invoice totalling fields
598 MOVETO "Invoice"
599 [Subtotal] = LineTotal ; update subtotal
600 IF (ISBLANK([Discount_%]) = True) ; discount amount
601 THEN [Discount_%] = 0
602 ENDIF
603 [Discount] = ROUND([Subtotal] * [Discount_%] / 100, 2)
604 IF (ISBLANK([Tax_%]) = True)
605 THEN [Tax_%] = 0
606 ENDIF
607 [Tax] = ROUND((([Subtotal] - [Discount]) * [Tax_%] / 100, 2) ; Tax Amount
608 IF (ISBLANK([Shipping]) = True)
609 THEN [Shipping] = 0
610 ENDIF
611 [Total] = ([Subtotal] - [Discount]) + [Tax] + [Shipping] ; Order Total
612
613 ENDPROC
614
615
616
617 ; -----
618 ; Procedure : PrintAllRecords()
619 ; Description:
620 ; The user wants to print a report on all Customer records. This
621 ; proc checks that the printer is ready and, if it is, then plays
622 ; a "pre-recorded" script to send the report to the printer.
623 ; Called By : Main()
624 ; -----
625
626 PROC PrintAllRecords()
627
628 ; Call the procedure IsPrinterReady() to check if a printer is on-line
629 ; and ready to receive the report. The IsPrinterReady() procedure returns
630 ; logical TRUE if the printer is ready and logical FALSE if it is not
631 ; ready. If the printer is not ready, then the PrintAllRecords()
632 ; procedure returns.

```

```

633 IF (IsPrinterReady() = False)
634     THEN RETURN
635 ENDIF
636
637 ; Open the currently set printer device for output. This command is
638 ; useful when working on a network.
639 OPEN PRINTER
640 ; RptAll.sc is a recorded script for printing the standard report
641 ; on the customer table.
642 Play "RptAll"
643
644 ; Close the currently opened printer.
645 CLOSE PRINTER
646 ENDPROC
647
648
649
650 ; -----
651 ; Procedure : PrintWestCoastRecords()
652 ; Description:
653 ; We only want West Coast, so we'll bring up a query save
654 ; we've previously done, run it, and copy the report to the answer
655 ; table, then print the report.
656 ; Called By : Main()
657 ; -----
658
659 PROC PrintWestCoastRecords()
660
661 ; Make sure that the printer is ready to receive information.
662 IF (IsPrinterReady() = False)
663     THEN RETURN
664 ENDIF
665
666 ; Open the Printer
667 OPEN PRINTER
668
669 ; The following query was created interactively and saved using
670 ; QuerySave. It was then copied into our script.
671 Query
672
673 Customer | Customer_No. | Last_Name | First_Name | Address | City |
674          | Check         | Check     | Check      | Check   | Check |
675
676 Customer | State/Prov. | Postal_Code | Telephone | Discount_% |
677          | Check       | Check 9..  | Check     | Check     |
678
679 Endquery
680
681 ; remember that the saved query ONLY places the query on the workspace.
682 ; It is still up to the programmer to execute the Do It! which will run
683 ; the query...
684 Do It!
685
686 ; the following can be produced by using ScriptRecord
687 ;Menu {Tools} {Copy} {Report} {DifferentTable}
688 ; {Customer} {R} {Answer} {R}
689 ;Menu {Report} {Output} {Answer} {R} {Printer}
690 ; But this can be done with PAL commands, and much more concisely.
691 COPYREPORT "Customer" "R" "Answer" "R"
692 REPORT "Answer" "R"
693
694 ; close the printer
695 CLOSE PRINTER
696
697 CLEARALL ; to clear query image and answer table
698

```

```

699  ENDPROC
700
701
702
703
704  ; -----
705  ; Procedure   : ShowSplashScreen
706  ; Description:
707  ;   Creates a window showing the application's "splash" screen.
708  ;
709  ; Called By   : Main()
710  ; -----
711
712  PROC ShowSplashScreen()
713
714  ; define the display attributes of the splash screen window
715  DYNARRAY WindowAttributes[]
716  WindowAttributes["ORIGINROW"] = 7
717  WindowAttributes["ORIGINCOL"] = 13
718  WindowAttributes["WIDTH"] = 56
719  WindowAttributes["HEIGHT"] = 11
720  WindowAttributes["STYLE"] = 30           ; yellow on blue
721  WindowAttributes["HASFRAEM"] = False   ; creates a frameless window
722  WindowAttributes["FLOATING"] = True     ; sets the window "above" the
723                                           ; echo layer
724
725  ; show the splash screen window
726  WINDOW CREATE ATTRIBUTES WindowAttributes TO SplashWindow
727
728  ; paint information inside the splash screen window
729  @ 2,0 ?? FORMAT("W56,AC", "B O R L A N D")
730  @ 3,0 ?? FORMAT("W56,AC", "Sample Application")
731
732
733  ; pause to show the window
734  SLEEP 2500
735
736  ; close the window
737  WINDOW CLOSE
738
739  ENDPROC
740
741
742
743  ; -----
744  ; Procedure   : IsPrinterReady
745  ; Description:
746  ;   Checks to see if the printer is ready.
747  ; Called By   : PrintAllRecords, PrintWestCoastRecords
748  ; -----
749
750  PROC IsPrinterReady()
751
752  ; check that the printer is ready
753  ; give user a message that we're checking the printer
754  MESSAGE "Checking printer status..."
755
756  ; PRINTERSTATUS() returns a value of True if the printer is ready,
757  ;   False otherwise.
758  IF (PRINTERSTATUS() <> True)
759  THEN BEEP
760  MESSAGE "Printer is not ready"
761  SLEEP 2500           ; pause to display message
762  MESSAGE ""          ; remove message
763  ; Tell the calling procedure NOT to continue sending the report.
764  RETURN False

```

```

765     ENDIF
766
767     MESSAGE "" ; remove "Checking..." message
768
769     ; Tell the calling procedure that the printer is ready to receive
770     ; report data.
771     RETURN True
772
773 ENDPROC
774
775
776 ; The Main Event
777 ; -----
778 ; The following line calls the Main() procedure. The Main() procedure
779 ; starts the Sample application.
780 Main()
781
782 ; Clean Up
783 ; -----
784 ; We're done. Close any open images and release any variables and procedures
785 ; before we exit.
786 RESET
787 RELEASE PROCS ALL
788 RELEASE VARS ALL

```

---

## Code listing for RPTALL.SC

```

1 : Menu {Report} {Output} {Customer} {R} {Printer}

```



# Glossary

For Paradox terms not specific to PAL, see the Glossary of the *User's Guide*.

- Abbreviated menu command** A group of PAL commands (such as ADD, PROTECT, SORT) that can be used to abbreviate a sequence of Paradox menu commands.
- Active** The most recently selected interface object. A window, dialog box, menu, or control can be active. *See also* Current.
- Application** A group of scripts, queries, or procedures forming a single unit, with which users can enter, view, maintain, and report on their data.
- Argument** Information passed to a command or function.
- Array** *See* Fixed array and Dynamic array.
- ASCII** An acronym for American Standard Code for Information Interchange; a sequence of seven-bit codes that define 128 standard characters, letters, numbers, and symbols. The IBM PC extends this code to 8 bits to include some special graphic characters. *See the PAL Reference* for a complete list of IBM ASCII codes.
- Binary field** A variable length field that can contain binary data only. You can create and restructure binary fields but not change their type. *See also* BLOB, Binary field, Variable length field.
- Blank** A field that contains no value; that is, a null field value. For numeric fields, a blank field is not equivalent to 0 unless specified in CCP with Blank=zero command.
- BLOB** An acronym for Binary Large Object. Paradox does not let you create BLOBs, but you can read them, store them in binary fields, and copy them to another binary field or a file. *See also* Binary field.

- Braced menu choice** A Paradox menu choice as represented in a recorded script; for example, {View} or {Tools}.
- Braces** The symbols { }, used to enclose Paradox menu choices in recorded scripts.
- Branching command** A control structure that performs specific commands depending on whether certain conditions are met.
- Canvas** The part of the screen that displays output to a user during the playing of a PAL script. When a script begins, the full-screen canvas is initialized with the state of the Paradox desktop. While the script is being played, the canvas hides the desktop unless the script explicitly calls for the user to view it. *See* Chapter 12 for a description of the PAL canvas and how it is used.
- Canvas cursor** The flashing underscore that points to the current position on the PAL canvas while a script is being played. Not the same as the workspace cursor. *See also* Workspace cursor.
- Checkmark** The symbol used in query statements to indicate that a field is to be displayed in the *Answer* table.
- Closed procedure** A procedure that exists as a self-contained unit, with limited ability to import and export the values of global variables and the definitions of procedures.
- Closed script** A script that is called from the operating system (at the DOS prompt, as in **C:\>paradox myscript**) and returns the user to DOS at its termination (ends with an EXIT command).
- Column** A vertical component of a Paradox table that contains one field. In a Paradox report, a vertical area containing one or more fields.
- Command** An instruction to PAL to perform a task. CLEAR and DELETE are examples of PAL commands. PAL commands perform an action; functions return a value. *See* the *PAL Reference* for a complete list of functions and commands. *See also* Branching command, Function.
- Compatible mode** A special mode that simulates the interface of Paradox 3.5 and lets you run existing applications. *See* Standard mode.
- Computed macro** A string expression that is evaluated and executed as a sequence of PAL commands with the PAL EXECUTE command. This lets you assign PAL commands to variables, which are later converted and executed.

<b>Concatenation</b>	The joining of two or more strings to form a single string. The PAL and Paradox concatenation operator is a + sign.
<b>Control structure</b>	A sequence of branching statements, such as IF... THEN... ENDIF or SWITCH... ENDSWITCH, that affects the order in which statements are executed in a script.
<b>Ctrl-Break</b>	In Paradox, a key sequence that stops the current task and returns you to the previous operation.
<b>Current</b>	The window, image, or canvas that will receive certain actions. The current window is the active, topmost window in the z-order. The current image is the Paradox object that was most recently active. The current canvas is the canvas that will display any canvas-writing commands. The current image, window, and canvas are not necessarily the same. <i>See</i> Chapter 11 for information about the current window and current image; <i>See</i> Chapter 12 for information about the current canvas. <i>See also</i> Active.
<b>Cursor</b>	A visual marker that indicates a location onscreen.
<b>Data</b>	A group of facts; in Paradox, the contents of individual fields and records in a table.
<b>Data type</b>	The type of data that a field, variable, or array element contains. PAL's available data types are similar to Paradox field types.
<b>Data value</b>	<i>See</i> Field value.
<b>Deadlock</b>	A situation created in a multiuser environment when two mutually exclusive lock commands are issued simultaneously.
<b>Debugger</b>	A built-in PAL facility that lets you interactively test and trace execution of commands in your scripts.
<b>Debugger cursor</b>	In the PAL Debugger, the indicator that points to the script command about to be executed.
<b>Debugger script line</b>	In the PAL Debugger, the line at the bottom of the screen that shows the script line being debugged.
<b>Debugger status line</b>	In the PAL Debugger, the line that indicates the name of the script being debugged and the current line number of the script.

- Desktop** The place in Paradox where windows reside. You do not have direct access to the workspace except through the desktop. *See also* Window, Workspace.
- Dialog box** An interface object that solicits specific types of information.
- Display image** In Paradox, a view of a table onscreen that shows the data in the table either in form or table view. *See also* Image, Query image, Form view, Table view.
- Dynamic array** A special kind of variable that consists of several separate elements. Elements in a dynamic array are designated by tags (also called subscripts) enclosed in square brackets, so that `Attributes["TITLE"]` and `Attributes[7/17/56]` are both elements in an array called *Attributes*. Values of different data types can be stored in different elements of the same PAL array. *See also* Fixed array, Dynamic array element.
- Dynamic array element** One component of an array. The number of elements in a dynamic array changes dynamically; unlike a fixed array, you do not specify the size when you declare a dynamic array. For example, the dynamic array created with the declaration `DYNARRAY Attributes[]` can have as many elements as will fit in memory. *See also* Fixed array element, Dynamic array.
- Dynamic scoping** A scheme in which the value of a variable in a procedure is allocated depending on whether that variable is global or private to the procedure.
- Editor** The component of Paradox used to create and edit scripts, text files, and memos. *See also* Memo field, Script.
- Encrypt** To translate a table or script into code that cannot be read without the proper password.
- Event** An individual packet of information that completely describes a specific, discrete occurrence at a specific time. Each user keystroke or mouse action generates an event; in addition, certain other conditions also generate an event. A PAL application can trap for triggers and events that occur in certain situations *See also* Trigger, Trapping.
- Example element** In a query statement, an arbitrary sequence of characters that stand for any value in a field. In Paradox, you indicate an example element by using *Example F5* to enter the characters in reverse video on the query form. In recorded query scripts, you indicate examples by preceding the characters with an underscore. Also called a linking element.

- Execute** To carry out a process, such as playing a script, adding new records to a table, deleting a file or record, and so on. The PAL EXECUTE command specifically evaluates a string expression as a sequence of commands to carry out.
- Expanded memory** Optional hardware not directly accessible to the processor; it always needs a software driver to function. You can add expanded memory to an 8086, 80286, or higher computer. Expanded memory allows an application to use up to 16MB of additional memory. Paradox prefers extended memory.
- Expression** A group of characters that can include data values, variables, arrays, operators, field specifiers, or functions that represent a quantity or value. A PAL expression can evaluate to a specific data type or, in certain cases, first be converted to string values before it is evaluated.
- Extended memory** Available only on 80286 or higher processor computers. Lets Paradox consider up to 16MB of memory to be regular memory. This far exceeds the conventional DOS limitation of 640KB. It also offers advantages over *expanded memory*. Extended memory can significantly enhance computing speed. *See* Chapter 23 of the *User's Guide* for more information on extended memory, expanded memory, protected mode, and real mode.
- Family** The entire set of objects related to a Paradox table, such as forms, reports, and indexes but not including the table itself. *See also* Object.
- Field** One type of information in a Paradox table. A collection of related fields makes up one record.
- Field assignment** Use of a field specifier to assign the value of an expression to a field.
- Field specifier** A special PAL construct that represents the value in a certain field. In its most basic form, a field specifier is simply a field name enclosed by square brackets, such as [Stock #]. Square brackets alone ([ ]) represent the value of the current field in the current record. *See* Chapter 3.
- Field type** The kind of information that can be contained in a field of a table. *See* Data type.
- Field value** The data contained in one field of a Paradox table. If there is no data present, the field is considered blank.
- Field view** A mode that lets the user move the cursor through a field character-by-character. It is used to view field values that are too large to be displayed in the current field width, or to edit a field

value. In Paradox, you can enter field view by pressing Field View *Alt-F5* or *Ctrl-F*. Entering field view in a memo field opens an Editor session. *See also* Editor, Memo field.

- File** A collection of information stored under one name on a disk. For example, Paradox tables and PAL scripts are stored in files.
- Fixed array** A special kind of variable that consists of several separate elements. Elements in a fixed array are designated by subscripts enclosed in square brackets, so that `a[1]` and `a[2]` are the first two elements of an array called *a*. Values of different data types can be stored in different elements of the same PAL array. *See also* Fixed array element, Dynamic array.
- Fixed array element** One component of an array. For example, the array created with the declaration `ARRAY a[7]` has seven array elements, `a[1]` through `a[7]`. *See also* Fixed array, Dynamic array element.
- Format specification** The way in which a field value is displayed onscreen or output to a printer. In PAL, you can use the `FORMAT()` function to control the appearance of fields. *See* Chapter 5.
- Form view** The representation of a table on a Paradox form, usually showing only one record at a time. A multi-record form can display more than one record at a time. A multi-table form can represent up to 10 tables at a time.
- Function** A built-in formula that performs computations or determines the status of PAL, Paradox, or your computer system. PAL functions include `ABS()`, `MIN()`, `MAX()`, `MOD()`, `TIME()`, `FIELD()`. PAL functions return a value; commands perform an action. You can use procedures to create your own functions in a script. *See the PAL Reference* for a complete list of functions and commands. *See also* Command, Procedure.
- Function keys** The ten keys located at the far left of the IBM PC keyboard, or across the top of the IBM AT Enhanced keyboard and the keyboards of some portable computers. Labeled F1 through F10.
- Global variable** A variable assigned in a PAL script that remains active throughout the entire script. *See* Private variable for the alternative.
- Help** You can press Help *F1* at any point in Paradox to display information about the current operation.
- IBM extended codes** Keys or combinations of keys on the keyboard that do not correspond to any of the standard ASCII character codes and are given special

“extended code” numbers between -1 and -132. These keycodes can be used anywhere in PAL where ASCII keycodes or Paradox key names can be used.

- Idle event** An event that is generated automatically at regular intervals by your system. *See also* Event.
- Image** A representation of a Paradox table or query form in the workspace. Images in the Paradox workspace are numbered counting from the top; the topmost image is 1, the next one down is 2, and so on. *See also* Display image, Query image.
- Image type** The type of an image in the Paradox workspace—either a display image (table) or a query image.
- Index** A special file that determines the location of a single record in a table quickly. *See also* Primary index, Secondary index.
- Key** A set of fields that uniquely identify each record in a table. *See also* Key field. Establishing a key has three effects: the table is prevented from containing duplicate records, the records are maintained in sorted order based on the key fields, and a primary index is created for the table. *See also* Primary index.
- Key event** An event that is generated by an individual key action such as pressing *Ctrl-F9*, *a*, or *Shift-F6*. *See also* Event.
- Keyboard macro** A single keystroke that causes Paradox to execute a specified series of keystrokes or commands.
- Keycode** A code that represents a keyboard character in PAL scripts. May be an ASCII number, an IBM extended keycode number, or a string representing a keyname known to Paradox. *See* Chapter 3 and the *PAL Reference* for more information.
- Key field** A field designated as all or part of an identifier of the records in a Paradox table. *See also* Key.
- Keypress interactions** The interactions between Paradox and the user, in which Paradox responds to keys pressed by the user.
- Keyword** A word reserved for use with certain specific commands. A keyword must not be used as the name of a variable, array, or procedure.
- Lifetime** The length of time an item is active. The lifetime of global variables is an entire Paradox session; once defined, they stay defined until you

leave Paradox or use `RELEASE VARS` to explicitly undefine them. The lifetime of private variables is the lifetime of the procedure in which they are defined; private variables are active only while the procedure is being executed.

- Link key** In a linked multi-table form, the part of the subordinate table's key that is linked or matched to fields in the master table.
- Link-locking** This happens when you begin to edit tables through a multi-table form. Paradox lets you make changes only through the form; if you toggle to table view and try to make changes, you'll get an error message.
- Logical operator** One of three operators—AND, OR, and NOT—that can be used on logical data. For example, an AND between two logical values results in a logical value of True if both the original values are also True.
- Logical value** A value (True or False) assigned to an expression when it is evaluated.
- Loops** Control structures that repeat a series of commands until a certain condition is met. *See* Control structures.
- Memo field** A variable length field that can contain any combination of characters and spaces. You can create and restructure memo fields. *See also* Binary field, Variable length field.
- Menu** A display of the choices or options available. Paradox menus appear at the top of the screen. Examples of menus are the Main menu displayed with *F10*, the PAL menu displayed with PAL Menu *Alt-F10*, and the Debugger menu displayed with PAL Menu *Alt-F10* if the Debugger is active.
- Menu area** The top line of the screen. The top line shows options; the status line at the bottom of the screen shows explanations. Both lines are used for status messages in some instances.
- Menu choice** A command chosen from a menu.
- Message** A string expression displayed in the message window.
- Message event** An event that is generated by your system upon attempting an interaction with a window or a `SHOWPULLDOWN` menu. *See also* Event.
- Message window** A window in the bottom right corner of the screen in which warnings, error messages, or other messages are displayed.



<b>Miniscript</b>	A selection on the PAL menu that lets you type in a sequence of commands of up to 175 characters. When you press <i>Enter</i> , the miniscript is executed. Miniscripts are not saved permanently.
<b>Mode</b>	The current Paradox system state—either Main, Edit, CoEdit, Restructure, Graph, Sort, Form, Report, Create, DataEntry, File Editor, Password, or Script. In addition, there are several minor modes that don't reflect the system state, such as field view and insert.
<b>Mode indicator</b>	An indicator in the upper right corner of the screen that shows the current Paradox mode.
<b>Mouse event</b>	An event that is generated by an individual mouse action such as a movement. <i>See also</i> Event.
<b>Normalized data structure</b>	An arrangement of data in tables where each record includes the fewest number of fields necessary to establish unique categories. Rather than using a few records to provide all possible information, a normalized table spreads out information over many records using fewer fields. A normalized table provides more flexibility in terms of analysis.
<b>Object</b>	A table, form, report, index, validity check file, or image setting file in a Paradox database. Together, a table and its related objects make up a family.
<b>Open script</b>	Any script that returns the user to Paradox at its termination.
<b>Parameter</b>	The variable into which an argument is passed. Used in defining procedures.
<b>Picture</b>	A pattern of characters that define what a user can type into a field during editing or data entry, or in response to a prompt.
<b>Primary index</b>	An index on the key fields of a Paradox table, used to determine the location of records, to keep them in sorted order, and to speed up operations.
<b>Private variable</b>	A variable (in a procedure) that has no meaning to the calling procedure or script.
<b>Procedure</b>	A program module consisting of a set of PAL statements that perform a specific task. <i>See also</i> Closed procedure.
<b>Procedure library</b>	A special file of up to 640 procedures. Procedures are stored in a library with the CREATELIB command and called from the library

into memory with the READLIB command. Once in memory, procedures are executed when called. Libraries have the file extension .LIB.

- Prompt** Instructions displayed on the screen. Prompts ask for information or guide the user through an operation.
- Protected mode** A feature of the microprocessor of a computer, available only on AT class machines with an 80286 or higher processor. Paradox runs in protected mode only.
- Prototyping** A process of application development in which small parts or the general structure of the application are designed and tested interactively using Paradox or the Workshop. These models are then used as the basis for building the finished system.
- QBE** *See* Query By Example.
- Query** A question asked of the data in a Paradox table, formulated in a query form.
- Query By Example (QBE)** Paradox's non-procedural query language that lets you ask questions about data by providing examples of the answers you are looking for.
- Query form** *See* Query image.
- Query image** An image of a table structure that is filled out with checkmarks and selection criteria to state a query. Also called a query form.
- QuerySave** An option on the Scripts menu that lets you save a query in script form.
- QuerySpeedup** An option on the Tools menu that lets you create a secondary index on a table.
- Query statement** One or more filled out query forms in the workspace.
- Quoted string** A representation of text typed outside the menu area, such as data entered into tables and literals on a form or report, in a recorded script. The text is enclosed in double quotation marks.
- Record** A horizontal row in a Paradox table that contains a group of related fields of data.
- Record number** A unique number that identifies each record in a table.

<b>Referential integrity</b>	A guarantee of internal consistency in your data even when the information about a single entity is stored in two or more tables.
<b>Relational database system</b>	As opposed to a flat-file database system, in which all related information is contained in a single table, a relational database system can store related information across a number of tables, and can create relationships between tables based on common data.
<b>Reserved words</b>	Names of commands, keywords, functions, system variables and operators. These words may not be used as PAL variable or array names.
<b>Restricted view</b>	A detail table on a multi-table form, linked to the master table on a one-to-one or one-to-many basis, limited to showing only those records that match the current master record.
<b>Row</b>	A horizontal component of a table, called a record in Paradox.
<b>Run-time error</b>	A script error that occurs when a valid command cannot be carried out in the current context.
<b>Script</b>	Any sequence of PAL statements stored in a file. A PAL script is equivalent to a program or database application. You can put statements in a script by recording Paradox keystrokes, or by typing them into the script using the PAL Editor (or other editor).
<b>Script Editor</b>	<i>See</i> Editor.
<b>Secondary index</b>	An index keeps track of and speeds up queries and other operations based on non-key fields in a table.
<b>Slash sequence</b>	A backslash followed by one or more characters, representing an ASCII character. Examples are \ <code>"</code> or \ <code>018</code> . Slash sequences are used for placing quotation marks within strings and including other characters that have special meaning to Paradox. <i>See</i> Chapter 3.
<b>SQL</b>	<i>See</i> Structured Query Language.
<b>Standard mode</b>	The new windowing interface of Paradox 4.0. <i>See</i> Compatible mode.
<b>String</b>	An alphanumeric value, or an expression consisting of alphanumeric characters.
<b>Structure</b>	In Paradox, the arrangement of fields in a table.

**Structured Query Language (SQL)** SQL has been adopted by the American National Standards Institute (ANSI) as the standard language for relational database management systems (RDBMS). SQL includes programming commands for data definition, data manipulation, data retrieval, security, and transaction processing.

**Subscript** A subscript is a way to reference an element in a fixed or dynamic array. Fixed array elements are referenced by numeric subscripts enclosed in square brackets; for example, `a[2]` refers to the second element of array *a*. Dynamic array elements are referenced by tags that can be any PAL expression; for example, `Attributes["TITLE"]` refers to the element of the `Attributes` array called *TITLE*.

**Substring** Any part of a string.

**Syntax error** A script error that occurs due to an incorrectly formed or expressed statement.

**System mode** *See* Mode.

**System state** The current condition of Paradox, or the mode that Paradox is in. *See also* Mode.

**Table view** The representation of a table in Paradox table format, in rows and columns.

**Tag** *See* Subscript.

**Termination commands** Control structures that return from procedures and leave scripts.

**Text Editor** *See* Editor.

**Tilde variable** A PAL variable used in a query form, which must be preceded by a tilde (~).

**Transaction** A group of related changes to a database.

**Transaction log** A list of changes made to a database that can be undone.

**Trapping** Using a PAL command to determine when a specific trigger or event occurs. You can use PAL to trap for different events and triggers and then take different actions accordingly. The commands `SHOWDIALOG`, `GETEVENT`, and `WAIT` let you take advantage of this feature of event-driven programming. *See also* Event, Trigger.

- Trigger** An artificial category of events that are available only to the PAL SHOWDIALOG and WAIT commands. Triggers represent very specific interactions during a WAIT or SHOWDIALOG command that are frequently useful for PAL applications to have access to. A trigger arises from a specific user action, independent of whether the action was caused by the keyboard or the mouse. The SHOWDIALOG and WAIT commands can trap triggers and call the *DialogProc* or *WaitProc* procedures. *See also* Event, Trapping.
- Validity check** A constraint or check on the values entered in a table.
- Variable** A place in memory to store data temporarily, for the length of the current Paradox session.
- Variable length field** A field with a fixed length section and an “overflow” section that changes size as necessary.
- Window** A viewing port that lets you look into a workspace object. *See also* Desktop, Object.
- Workspace** The place in Paradox where objects, such as tables, forms, and queries reside. The workspace is the area where all Paradox operations take place. *See also* Desktop, Object, Window.
- Workspace cursor** The marker in the Paradox workspace that points to the current position in a field, record, or other object. *See also* Canvas cursor.



# match character, 79, 82  
 ( ) (parentheses)  
   expressions and, 49, 50  
   PAL functions and, 21, 54  
 \* match character (repetition), 80  
 \* operator (multiplication), 48  
 + operator (addition/concatenation), 47, 54  
 - operator (negation/subtraction), 47  
 .. operator (wildcard), 57  
 / operator (division), 48  
 / separator in date fields, 85  
 < operator (comparison), 48  
 <> operator (comparison), 49, 50, 54  
 = command, 40, 49  
 = operator (comparison), 49, 54  
 > operator (comparison), 48  
 ? command, 193, 194  
 ?? command, 193, 194  
 @ command, 192  
 @ operator (wildcard), 57  
 \_ (underscore) in query examples, 27  
 { } match character (inhibit), 84  
 { } match character (set), 80  
 ~ (tilde) variables, in queries, 41, 343  
 ~ labels, in scripts, 244, 261

## A

abandoning script play, 160  
 abbreviated menu commands, 27–30, 144, 323  
 ACCEPT, 77, 213, 214, 378  
   REQUIRED keyword and, 78  
   Retval and, 42  
 ACCEPTDIALOG, 279  
 active window, 181  
 addition operator, 47, 54  
 alignment specifications, 87  
 alphanumeric (A) data types, 32  
   comparing, 49  
   returning, 34  
 alphanumeric (A) field types, 32  
 alphanumeric constants, 34–35  
 alphanumeric fields  
   comparing with memo fields, 49

  currency (\$) data types in, 87  
   sorting, 49  
   truncated, 87  
 AND operator, 50  
 APPENDARRAY, 45, 327  
 application layer, 182  
 applications, 1–6, 15  
   accounting, 88  
   building, 6, 325, 335, 365–374  
   creating subroutines for, 20, 93  
   creating, 366–371  
   data entry, 330, 397  
   distributing, 372–374  
     *See also* Paradox Runtime  
   documenting, 374  
   editing, 370  
   large, 105, 106, 107, 139, 371, 374  
   memory management and, 355–358  
   modular, 76, 100, 105, 335  
   multi-table forms and, 303–314  
   order entry, 398  
   performance, facilitating, 353–361  
   problems with running, 120  
   protecting, 107  
   running, 349, 354, 356, 373, 374  
   running DOS programs from, 125  
   stepping through, 370  
   terminating, 105  
   testing, 370–371  
   user interfaces and, 169  
   version compatibility, 8  
 applications, multiuser  
   *See* multiuser applications  
 applications, sample  
   *See* Sample application  
 arbitrary keypresses, 23  
 area codes, entering in fields, 82  
 arguments, 21, 395  
   debugging, 160  
   PAL commands and functions, 21, 31  
   procedures, 94–97  
 ARRAY, 43  
 array elements, 240  
   assigning, 338

- debugging, 157
- declaring private, 100
- dynamic vs. fixed, 45
- passing in procedures, 100, 102
- reassigning, 44
- referencing, 43, 45, 338, 426
- saving, 119
- storing data types in, 43, 44, 337
- tags and, 45, 46
- testing, 132
- array indexes, 43
  - color palette, 196, 200
  - windows, 173, 175
- arrays, 43–46, 337–341
  - allocating memory for, 43
  - assigning values to, 43–46
  - creating, 338
  - declaring private, 102
  - naming, 38
  - passing as parameters, 100
  - passing to procedures, 102, 226
  - releasing, 44, 105
  - returning size of, 44, 46
  - working with records in, 45, 313, 327, 337
  - See also* dynamic arrays; fixed arrays
- ARRAYSIZE(), 44
- ASC(), 61
- ASCII characters, 34, 58
  - entering in scripts, 35
- ASCII codes, 35
  - entering in expressions, 58, 59
- ASCII text editors
  - See* text editors
- assignment, 40, 49
- asterisks, in fields, 87
- attached canvases
  - See* image windows
- attributes
  - canvas, 194–195
  - color, 194
  - dialog boxes, 238, 241
  - system, 195–201
  - window, 174–177
- Autolib system variable, 108, 110
- autoload libraries, 110
- automatic fill-in, pictures, 78, 79
  - inhibiting, 82, 83, 84
- automatic locks, 379, 385
- automatic retry period, 393
  - disabling, 393
  - setting, 393
- autorefresh, 385

- auxiliary passwords, 116
  - See also* passwords

## B

- background colors, setting, 195
- background elements, dialog boxes
  - See* canvas elements
- backslash sequences, 35, 58
- batch files, 125, 374
- BeginRecord command
  - PAL, 130, 141
  - Scripts, 141
- blank lines, 18
- blank strings, 38
- blank values, 38, 49
  - assigning to variables, 40
  - multi-table forms and, 314
  - testing for, 38
- Blank=Zero command (CCP), 38
- BLANKDATE(), 38
- BLANKNUM(), 38
- blanks, treating as zeros, 38
- Boolean values, 33
  - See also* logical values
- branching commands, 64–69
  - loops and, 69
  - See also* control structures
- branching functions, 69
- built-in calculator, 131, 156

## C

- cache, 354–356
  - central memory pool and, 357
  - swap devices and, 356
  - See also* memory
- calculated fields, formatting, 37
- calculation functions, 330–332
- calculator, built-in, 131, 156
- call chain, 153, 160
- CANCELIALOG, 280
- canvas, 17, 171, 188, 191
  - attributes, setting, 194–195
  - building incrementally, 191
  - coordinates, 191
  - current, specifying, 188, 189
  - cursor, 191–192
  - default, 186
  - retrieving handle for, 190
  - storing, 190
  - types, 185–186



- writing to, 185–204
- canvas elements, 262
  - See also* dialog boxes
- canvas windows
  - See* windows
- CanvasElements parameter, 242, 262
- case sensitivity
  - keycodes, 23
  - keywords, 18
  - menu commands in scripts, 23
- case specifications, 87
- central memory pool, 357–358
- Changes command (Tools), 394
- character strings, 58
  - See also* strings
- characters
  - control, 35
  - reading from keyboard, 213
  - repeating, in fields, 80–82
  - tab, 35
- characters, ASCII
  - See* ASCII characters
- CHECK, 27
- check boxes, 255–257
  - creating, 256
  - radio buttons vs., 253
- checkmarks
  - entering in query statements, 344
  - reproducing, 27
- CLEARALL, 117, 328
- CLEARPULLDOWN, 236
- closed procedures, 105–107, 335
  - calling, 107
  - defining, 107
  - nesting, 105
  - terminating, 105
- code pool, 357–358
  - See also* memory
- CoEdit mode
  - activating, 390
  - multiuser applications and, 376
  - record locking and, 385
- COEDITKEY, 390
- COL(), 192
- color
  - attributes, 194–204
  - monitors, 194
  - palette, 194, 195–204
- comma, as decimal point, 37
- command categories, 20
  - abbreviated menu commands, 27–30
  - keypress interactions, 22–27
  - programming, 21
  - See also* specific command category
- commands, 15, 17–30
  - arguments in, 31
  - branching, 64–69
  - case sensitivity, keywords in, 18
  - commenting, 18
  - entering in scripts, 17
  - executing, 19–20, 134, 318, 347
  - functions vs., 21
  - indenting, 18
  - keyboard macros and, 348
  - miniscripts and, 133
  - procedures, 91
  - repeating sequence of, 69
  - restricted views and, 308
  - syntax, 15–18
  - terminating, 75–76
  - testing, 135
  - tracing through, 158, 164
  - See also* specific command
- commands, menu
  - See* menu commands
- comments, entering in programs, 18
- comparison operators, 48–49, 50
  - blank values and, 49
  - using with strings, 54
- compatible mode, 9
- concatenation, 47, 54–55
- conditions, 63–64
  - branching commands and, 64
  - loops and, 69, 72
  - testing, 66
  - See also* control structures
- constants, 34–38
  - alphanumeric, 34–35
  - date, 37
  - logical, 38
  - rtumeric, 35–37
- continuous play, scripts, 130, 158
- control characters, 35, 59
- control elements, 242
  - check boxes, 255–257
  - default, activating, 261
  - moving to specific, 274
  - pick lists, 272, 244–253
  - push buttons, 243–244
  - radio buttons, 253–255
  - sliders, 264, 259–261, 269
  - synchronizing, 264, 269–274
  - type-in boxes, 258–259, 264, 269, 272
  - updating, 264, 269, 275–279

*See also* dialog boxes  
 control structures, 19, 63–76  
   branching commands in, 64–69  
   conditions, testing, 63–64, 66  
   loops, 69–75  
   multiple branching in, 66, 68  
   nesting, 64, 65  
   terminating commands in, 75–76  
 ControlElements parameter, 242  
 CONVERTLIB, 114  
 coordinates  
   canvas, 191  
   mouse events, 210  
   window, 172  
 COPY, 379  
 COPYFROMARRAY, 45, 313, 327, 338, 339  
 COPYTOARRAY, 45, 327, 338  
 CREATE, 321  
 CREATELIB, 108  
   SIZE keyword and, 108  
 Ctrl  
   *See* control characters  
 Ctrl-Break, 149  
   problems with, 115  
 currency (\$) data types, 32  
   entering in alphanumeric fields, 87  
   returning, 36  
 current canvas, 189  
   specifying, 188, 189  
 current field, 51  
 current image, 178, 189  
 current mode, 319  
 current window, 189  
   specifying, 178, 181  
 cursor, 192  
   canvas, 191–192  
   Debugger, 154  
   moving in dialog boxes, 274  
 cursor keypad  
   *See* keypad

## D

data, 370  
   changing, 186, 188  
   coediting, 381, 385, 392  
   editing, 26  
   linking, 304, 312  
   printing, 86  
   restricting access to, 119, 307  
   storing, 97  
   working with, 333

data entry, 318, 367  
   alternatives, 82  
   automatic, 77, 342  
   facilitating, 77, 350  
   mandatory, 79, 82  
   multi-table forms, 314  
   multiuser applications, 376, 378–392, 394  
   optional, 81  
   repeating characters, 81  
   restricting, 378, 379  
   testing, 370  
 data entry applications, 330, 397  
 data integrity, multiuser applications, 375–379  
 data types  
   array elements and, 43, 44, 337  
   assigning to fields, 52, 85  
   automatic conversion and, 336  
   combining, 33, 47, 132  
   comparing, 48, 49  
   formatting, 85–89  
   losing, 44  
   returning, 32–38, 336  
   storing, 44, 45  
   variables and, 38, 40, 336  
   *See also* specific type  
 database applications  
   *See* applications  
 date  
   constants, 37  
   fields, 83  
   fields, asterisks in, 87  
   formats, 37, 55, 89  
   specifications, 89  
 date (D) data types, 32  
   returning, 37, 55  
 date functions, 57  
   *See also* specific date function  
 dates  
   adding, 47  
   aligning in fields, 87  
   blank, returning, 38  
   comparing, 56  
   converting to strings, 55  
   entering in expressions, 55–57  
   formatting, 37, 56, 89  
   returning, 47, 101  
   searching for, 57  
   subtracting, 47  
   valid, 37  
 DATEVAL(), 55  
 deadlock, 383  
 DEBUG, 152, 161

- Debug command
    - PAL, 131, 152
    - Script Break, 152
  - Debugger, 114, 151–167, 370
    - activating, 131, 152
    - call chain and, 153, 160
    - miniscripts and, 136
    - procedure libraries and, 154
    - quitting, 161–162
    - shortcut keys, 155
    - tutorial for, 162–167
  - Debugger menu, 156
    - activating, 156
  - Debugger screen, 154
    - cursor in, 154
  - Debugst script, 162
  - decimal places, 86
  - decimal point characters, 37
  - delays, 135
  - desktop
    - layer, 181
    - updating, 186
    - working with, 327–330
    - writing to, 186–187
    - See also* workspace
  - detail forms, 303–304
    - embedding in master forms, 298, 303, 305–307
    - linking to master forms, 304, 307
    - See also* multi-table forms
  - detail records
    - changed values in, 313
    - hierarchy, 305
    - locking, 391, 392
    - moving to, 311
  - detail tables, 304
    - key fields in, 305, 312–314
  - dialog boxes, 237–284
    - accepting, 279
    - adding titles to, 242
    - attributes, 202, 238, 241
    - canceling, 280
    - canvas elements in, 262
    - control elements in, 242–261
    - creating, 239, 241, 243
    - cursor, moving through, 274
    - display attributes, 238, 241
    - exiting, 243, 279
    - incorporating into applications, 368
    - interactions, controlling, 267–284
    - keypresses and, 244, 258, 261
    - modal, 238
    - mutually exclusive choices in, 253
    - procedures, terminating, 265
    - related options in, 255
    - scrolling, 259
    - synchronizing elements in, 264, 269–274
    - updating elements in, 264, 268, 269, 275–279
    - version compatibility, 238
  - DialogProc procedure, 263–267
    - arguments for, 264
    - calling, 212, 264
  - directories
    - accessing in shared environments, 376, 377
  - directory locks, 380–381
  - directory paths
    - networks, 377
    - setting in pick lists, 245, 246
  - discrete occurrences, 205
  - display attributes
    - See* video attributes
  - display formats
    - See* format specifications
  - distributing applications, 372
  - division operator, 48
  - Do-It (F2), 23
  - Do-It!, 318
  - documentation, 7–12
    - applications and, 374
    - printing conventions, 8
  - DOS, 372
    - playing scripts from, 148
    - returning to, from scripts, 19
    - shelling out to, 125
    - wildcards, 239, 245
  - dynamic array (DY) data types, 32
    - returning, 46
  - dynamic arrays, 43, 340–341
    - branching with, 68
    - building menus with, 68, 226, 230
    - creating dialog box elements with, 245, 250, 252
    - declaring, 45–46
    - events and, 209, 210, 216
    - returning size of, 46
    - stepping through, 71
    - window attributes and, 174, 175, 176
    - See also* arrays; tags
  - dynamic scoping
    - See* scope
  - DYNARRAYSIZE(), 46
- ## E
- ECHO, 19, 181, 186–188
    - disabling, 186

- layer, 181
- Edit mode, 376
- edit specifications, 87
  - using with sign specifications, 88
- Editor, 144–145
  - activating, 145
- Editor (Ctrl-E), 161
- Editor command
  - Debugger, 161
  - Scripts, 145
- editors
  - built-in, 144
  - text, 16, 144, 351
- empty strings, 38
- encrypted scripts, editing, 145
- encryption
  - See* passwords
- End-Record (Alt-F3), 131
- End-Record command
  - PAL, 131
  - Scripts, 141
- error-handling procedures, 123–125
  - defining, 124
  - executing, 121
  - return codes, 124
  - trapping for low memory conditions, 358
- ERRORCODE(), 124
  - low memory conditions and, 358
  - record locks and, 386, 387, 393
  - table locks and, 382, 393
- ERRORINFO, 124
  - record locks and, 386
- ERRORMESSAGE(), 124
  - record locks and, 387
  - table locks and, 382
- Errorproc system variable, 124
- errors, 120–125, 386
  - codes, 121, 124
  - detecting, 164
  - low memory, 357–358
  - messages, Debugger, 154
  - messages, returning, 124
  - procedures, 358
  - returning information on, 124
  - testing for, 125
  - trapping, 121, 152
- ERRORUSER(), 124, 382
- event
  - categories, 205–207
  - stream, 207–209, 214
  - tags, 216–218
- event-driven programming, 205–221, 237
- EventList parameter, 209
  - SHOWDIALOG and, 242, 263
  - WAIT and, 211
- events, 205–207
  - attributes, returning, 209
  - blocking, 213–214
  - denying, 220
  - dialog boxes and, 212, 264, 265
  - executing, 209, 216–218
  - filtering, 210, 213
  - processing, 207–209, 214–215
  - queued, 207
  - trapping, 209–214, 264
  - triggers and, 218, 220
  - valid, 216
  - WAIT procedure and, 289
  - See also* specific event category
- example elements
  - queries and, 343
  - reproducing, 27
- EXECEVENT, 209, 216
- EXECPROC, 65, 157
- EXECUTE, 157
- Execute script, 119
- EXIT, 19, 20, 76
- expanded memory, 355
  - as swap device, 356
- explicit locks, 379, 385
- exponential notation, 36
- expressions, 31–61
  - arrays in, 43–46
  - blank values in, 38
  - constants in, 34–38
  - data types, returning in, 32–38
  - debugging, 157
  - editing, 132
  - evaluating, 33, 50, 131, 156
  - field specifiers in, 50–53, 326, 333–334
  - functions vs., 53–54
  - keycodes in, 58–61
  - operators, 46
  - procedures and, 96
  - referencing fields in, 51
  - subscript, 44
  - testing, 33, 131, 156
  - variables in, 38–43, 334–337
  - wildcard operators and, 57
- extended memory, 355

## F

- family, 303
- FIELD keyword, 285
- field names, in arrays, 45, 339
- field specifiers, 50–53, 326, 333–334
  - entering in expressions, 51
- field types, 32, 77, 78
  - date, 83
  - See also* specific field type
- field values
  - transferring, 338
  - See also* values
- field view, 320
- fields, 77
  - assigning values, 52, 327
  - current, 51
  - logical, simulating, 33
  - moving to specific, 53
  - referencing, 50
  - width, setting, 86
  - See also* key fields
- file names, displaying in pick lists, 239, 245
- file-name filters, 239, 245
- files, 238, 245
  - batch, 125, 374
  - lock, 377, 381
  - read-only, 377
- FILEVERSION(), 113
- fixed array (AY) data types, 32
  - returning, 44
- fixed arrays, 43, 337–339
  - building dialog boxes with, 240
  - creating dialog box elements with, 245, 248
  - declaring, 43–45
  - stepping through, 71
  - See also* arrays
- FOR, 70–71
  - ENDFOR keyword and, 70
  - FROM keyword and, 70
  - STEP keyword and, 70, 71
  - TO keyword and, 70
- FOREACH, 71–72
- foreground colors, setting, 195
- form locks, 392
- form view, 320
  - activating, 310
- format specifications, 85–89
- FORMAT(), 56, 85, 193
- FORMKEY, 310
- forms, 367

- calculated fields and, 37
  - multi-page, 303
  - multi-record, 303
  - multi-table, 303–314
  - returning status of, 304
- FORMTABLES, 304
- FORMTYPE(), 304
- full locks, 379, 380, 381
- full-screen canvas, 186, 188
  - restoring, 188
- function keys, entering in expressions, 59
- functions, 53–54
  - arguments in, 21, 31
  - branching, 69
  - calling, 53
  - commands vs., 21
  - international number formats and, 37
  - procedures and, 91
  - restricted views and, 308
  - user-defined, 53
  - See also* specific function

## G

- GETCANVAS(), 190
- GETCHAR(), 61, 135, 136, 213, 342
- GETCOLORS, 195
- GETEVENT, 206, 209–210
  - building menus with, 215–233
  - EventList parameter and, 209
- GETMENSELECTION, 229, 230
- GETWINDOW(), 173
- global colors, 195–201
- global echo, 186
- global variables, 41
  - changing, 99, 105
  - dynamic scoping of, 100, 103, 335
  - procedures and, 99
- Go (Ctrl-G), 158
- Go command (Debugger), 158
- group locks, 391

## H

- handles
  - window, 172–174, 190
  - See also* window handles
- handles, window, 172–174
- Help mode, 320
- hierarchical structure, multi-table forms, 305–306
- highlighting menu commands, 224

## I

- I/O, facilitating, 382
  - See also* input; output
- IBM extended codes, 60
- idle events, 207, 269, 290
  - trapping, 209
  - valid tags for, 218
  - See also* events
- IF, 64, 65–66, 69
  - ELSE keyword and, 65
  - ENDIF keyword and, 65
- IIF(), 69
- image windows, 186, 187
  - See also* canvas
- images
  - calculating values in, 331
  - moving to, 52, 53
  - referencing fields in, 50
  - restricted views and, 307
  - specifying active, 52
- INDEX, 359, 360
  - MAINTAINED keyword and, 360
- indexes, 359–361, 367
  - building, 359
  - maintaining, 360–361
  - primary, 359
  - regenerating, 360
  - secondary, 359, 360, 361
  - See also* array indexes
- INFOLIB, 112
- Init script, 119–120
  - keyboard macros in, 135, 348, 352
- input, 77–84, 370
  - accepting from keyboard, 214
- insert mode, 319
- Instant script, 119, 120
  - losing, 140
  - playing, 148
  - recording, 140
  - renaming, 140
- Instant Script Play (Alt-F4), 120, 148
- Instant Script Record (Alt-F3), 120, 140
- INSTANT.SC, 120
- international number formats, 36
- interruptions, 149
  - problems with, 115
- ISASSIGNED(), 40
- ISBLANK(), 38
- ISBLANKZERO(), 38
- ISMULTIFORM, 304
- ISTABLE(), 30, 321
- ISWINDOW(), 174

## K

- key conflicts
  - See* key violations
- key events, 206, 290
  - processing, 208
  - trapping, 209, 213–214
  - valid tags for, 217
  - See also* events
- key fields
  - deleting, 360
  - in detail tables, 305, 312–314
  - in master tables, 314
  - indexes and, 359
  - returning information on, 389
- key names, entering in expressions, 59
- key violations, 388
  - resolving, 389
- keyboard, 58
  - accepting input from, 214
  - keyboard buffer, reading from, 213
  - keyboard macros, 134, 139, 347–352
    - canceling, 352
    - creating, 347–349, 351
    - keycodes in, 348
- keycodes, 23
  - assigning to keyboard macros, 348
  - converting to ASCII numbers, 61
  - entering in expressions, 58–61
  - returning, 60, 61, 213
- KEYLOOKUP, 389
- KEYPRESS, 23, 324, 342
- keypress interactions, 22–27, 320–324
  - simulating, 323–324
- keypresses, 58, 205
  - arbitrary, 23
  - automating, 120
  - dialog boxes and, 244, 258, 261
  - recording, 130, 139, 140, 147, 320
  - simulating, 22–23, 323–324
  - trapping for, 61
- keys
  - Debugger, 155
  - redefining, 134
  - referencing in expressions, 58–60
- keywords, 21
  - case sensitivity, 18
  - using in scripts, 39

## L

- labels, 261–262
  - creating, 261

- See also* keypresses, dialog boxes and
- large applications, 105, 106, 107, 139, 371
  - documenting, 374
- large procedures, 354
- large tables, 360
- .LCK files, 377
- libraries, 378
  - accessing, from shared environments, 377
  - allocating space for, 108
  - autoload, 107, 110
  - creating, 108
  - debugging, 154
  - including with applications, 372
  - listing procedures in, 112
  - naming, 372
  - procedure, 107–114
  - reading, 108, 110
  - rebuilding, 109, 113
  - returning information on, 112, 113
  - size, reducing, 113
  - upgrading, 114
  - version compatibility, 113
- lines, blank, in programs, 18
- link keys, 305
  - assigning values to, 312, 313
  - changing, 313
  - updating, 313
  - See also* multi-table forms
- link type, returning, 306
- link-locking, 313
- links, 304–310, 313
- LINKTYPE(), 306
- list boxes
  - See* pick lists
- literal characters
  - pictures, 78–80, 82, 83
  - See also* picture strings
- literals, scripts and, 26
- local colors, 201–204
- local echo, 186
- LOCALIZEEVENT, 210
- LOCATE, 325, 359, 390
  - Retval and, 42
- LOCATE INDEXORDER, 325
- LOCK, 382, 383
  - Retval and, 42
- Lock command (Tools), 384
- lock files, 377, 381
- Lock Toggle (Alt-L), 386
- lock types, 380–381
  - combining, 384
  - multi-table forms, 392
- See also* specific lock type
- LOCKKEY, 386
  - Retval and, 42
- LOCKRECORD, 385–387, 388, 391
  - Retval and, 42
- locks, 376, 379, 394
  - automatic, 379, 385
  - directory, 380–381
  - explicit, 379, 385
  - failing, 382
  - group, 391
  - key violations and, 389
  - multi-table forms and, 391, 392
  - multiple, 384, 390
  - record, 381, 385–392
  - releasing, 383, 384, 391
  - Retval and, 42
  - status, returning, 384, 387
  - table, 379–384
  - testing, 376, 383, 393
  - write-record, 392
  - write, 379–381
  - See also* record locks; table locks
- LOCKSTATUS(), 384
  - ANY keyword and, 384
- logic errors, 121
- logical
  - constants, 38
  - fields, simulating, 33
  - operators, 49
  - specifications, 89
- logical (L) data types, 32, 33
  - returning, 38
- logical values, 33
  - comparing, 48
  - default alignment, 87
  - formatting, 89
  - negating, 50
  - picture strings and, 83
  - returning, 42, 49, 63
  - truncated, 87
- Lookup (Alt-K), 389
- LOOP, 74
- loops, 69–75
  - executing indefinitely, 70
  - nesting, 69
  - quitting, 74
  - repeating, 74
  - stepping through, 70
  - See also* control structures
- low memory errors, 357–358
- lowercase characters, assigning to strings, 87

## M

macros, 107, 133, 348

  autoloading, 348

  canceling, 352

  keyboard, 134, 139, 347–352

*See also* keyboard macros

Main mode, 318

many-to-many relationships, 306

many-to-one relationships, 306

master forms, 303

  embedding detail forms in, 298, 303, 305–307

*See also* multi-table forms

master passwords

  defining, 116

*See also* passwords

master records

  changing, 312, 391

  deleting, 313

  hierarchy, 305

  locking, 392

  moving to, 311

master tables, 304

  key fields in, 314

*See also* multi-table forms

match characters

  pictures, 78–84

  repeating group of, 80

*See also* picture strings

memo (M) data types, 32

  returning, 34

memo (M) field types, 32

memo fields, comparing with alphanumeric, 49

memory

  allocating, 43, 356

  applications and, 355–358

  cache, 354–356

  central memory pool, 357

  code pool, 357–358

  expanded vs. extended, 355

  freeing, 41, 105, 107

  low, error conditions, 357

  procedures and, 92, 106, 110, 354–355

  returning information on, 358

  scripts and, 353–354

memory manager

*See* VROOMM

menu bars, building, 229

menu commands, 215

  abbreviated, 27–30, 144, 323

  automating, 120

  choosing, 22, 28, 224

  default, specifying, 224

  disabling, 235

  entering in scripts, 144

  highlighting, 224

  recording, 22, 139

*See also* menus

MENUDISABLE, 235–326

MENUENABLE, 235–236

menus

  attributes, 227

  branching commands and, 65, 68

  building, 230

  display attributes, 227

  incorporating into applications, 368, 369

  modal, 224

  nesting, 236

  non-modal, 224

  pop-up, 223–228

  pull-down, 215, 223, 229–236

  redisplaying, 72

  returning control to, 72

  stacking, 236

  version compatibility, 223

*See also* menu commands

MESSAGE command, 287

message events, 206, 290

  ignored, 208

  mouse interactions and, 208

  processing, 208

  trapping, 207, 209

  valid tags for, 217

*See also* events

messages, 135, 192–193

  displaying onscreen, 125, 187, 214

  system-generated, 125

*See also* error messages

Miniscript command

  Debugger, 159

  PAL, 133, 134, 136, 159

miniscripts, 119, 133–136, 159

  building, 133

  changing, 153

  editing, 136

  playing, 159

  writing to screen, 135

*See also* scripts

modal

  hierarchy, 318

  menus, 224

modal dialog boxes, 238

*See also* dialog boxes

modes

  minor, 319



- returning current, 28, 319
- See also* specific mode
- modes, system
  - See* system modes
- modular applications, 76, 100, 105, 335
- MONITOR, 194
- monitors, 194
- mouse, 207
- mouse events, 206, 290
  - coordinates, returning, 210
  - ignored, 207
  - processing, 207
  - recording, 140
  - trapping, 209, 210
  - valid tags for, 216
  - See also* events
- mouse interactions, 205, 213
  - message events and, 208
- MOVETO, 53, 325
- multi-line
  - comments, 18
  - messages, 192, 193
- multi-page forms, 303
- multi-record forms, 303
- multi-table forms, 303–314
  - accessing, 296, 313, 314
  - blank values in, 314
  - creating, 303, 305
  - entering data in, 314
  - hierarchical structure of, 305–306
  - linking tables to, 304–310, 313
  - locking, 391–392
  - moving between tables in, 296
  - networks and, 391–392 opening, 310
  - password-protecting, 314
  - querying, 309
  - Sample application, 401
  - working with, 310–312
- multiplication operator, 48
- multiuser applications, 375–396
  - adding records to, 387
  - building, 376–378
  - changing records in, 385, 387, 393–395
  - converting to, 376
  - data integrity and, 375–379
  - declaring tables private to, 377
  - developing, 368
  - locking shared objects and, 376, 379–396
  - referential integrity and, 391, 392
  - refreshing, 385
  - setting access levels for, 378, 379
  - testing, 376

## N

- naming conventions
  - script elements, 38
- negation operator, 47
- negative numbers, 60
  - formatting, 88
- networks
  - accessing tables on, 117, 119
  - building applications for, 375–396
  - creating private directories for, 377
  - data entry and, 376, 378–392, 394
  - debugging applications for, 376
  - debugging tutorial, 162
  - error trapping and, 123, 124
  - key violations and, 388
  - locking tables, 379–393
  - multi-table forms and, 391–392
  - password-protecting tables, 378
  - read-only files and, 377
  - running applications on, 372
  - storing objects on, 376–378
  - See also* multiuser applications
- NEWDIALOGSPEC, 281
- NEWWAITSPEC, 299
- Next (Ctrl-F4), 181
- Next (Ctrl-N), 158
- Next command (Debugger), 158
- non-modal menus, 224
- nonsensical data, 370
- NOT operator, 50
- number formats, international, 36
- numbers
  - aligning in fields, 87
  - blank, returning, 38
  - converting to strings, 55
  - entering in scripts, 36–37
  - formatting, 36, 86, 87–88
  - negative, 60
  - picture strings and, 83
  - positive, 60
  - repeating, in fields, 81
  - sequential, 330
- numeric (N) data types, 32
  - returning, 36, 55
- numeric constants, 35–37
  - entering in expressions, 36
- numeric fields
  - asterisks in, 87
  - leaving blank, 38
- numeric keypad
  - See* keypad
- NUMVAL(), 55

## O

- objects, 171
  - accessing, 376, 377, 378
  - annotating, 186
  - developing applications and, 373
  - duplicating names of, in scripts, 39
  - echoing to screen, 19
  - locking, in multiuser applications, 376, 379–393
  - memory allocation and, 355, 356
  - password protecting, 116
  - stacking, 179–182
  - unlocking, 384
- one-to-many relationships, 305
  - checking, 312
- one-to-one relationships, 305
  - checking, 312
- operators, 46
  - comparison, 48–50, 54
  - logical, 49
  - precedence, order of, 50
  - wildcard, 57, 344
  - See also* specific operator
- OR operator, 50
- order of execution, commands, 19–20
- output, 85–89
  - aligning, 87
  - echoing, 19, 182, 186–188
  - facilitating, 350
  - formatting, 185–193
  - scripts, displaying onscreen, 135
  - writing to screen, 350
- overwrite mode, 319
- owner passwords
  - See* master passwords

## P

- PAINTCANVAS, 194, 195
- PAL, 1, 2–7, 365
- PAL canvases
  - See* canvas
- PAL commands
  - See* commands
- PAL Debugger
  - See* Debugger
- PAL Editor
  - See* Editor
- PAL functions
  - See* functions
- PAL menu, 129
  - activating, 129
  - exiting, 137

- palette
  - See* color palette
- PALMenu (Alt-F10), 129, 141
- Paradox
  - customizing, 120
  - problems with, 351
  - using earlier versions of, 8, 113, 373
  - using interactively, 319, 320, 365, 367, 371
- Paradox Application Language
  - See* PAL
- Paradox Runtime, 5, 148, 373
- PARADOX.LCK, 381
- PARADOX.LIB, 109
- parameters, 21
  - explicitly passing, 395
  - See also* arguments
- parentheses
  - expressions and, 49, 50
  - PAL functions and, 21, 54
- PASSWORD, 116, 378
- Password command (Tools), 378
- passwords, 115–119, 258
  - assigning to multi-table forms, 313, 314
  - auxiliary, 116
  - clearing, 383
  - master, 116
  - prompting for, 378
  - releasing, 116, 117
  - returning, 378
- pattern operators
  - See* wildcard operators
- patterns
  - See* picture strings
- PDOXUSRS.LCK, 381
- performance, facilitating, 353–361
- pick lists, 244–253
  - creating, 238, 240, 245, 247, 248, 250, 252
  - directory paths, specifying, 245, 246
  - scroll bars in, 259
  - synchronizing, 272
- PICKFORM, 310
- picture strings, 77–84
  - alternative entries in, 82
  - editing, 78
  - filling automatically, 78, 79
  - inhibiting automatic fill-in, 82, 83, 84
  - optional entries in, 81
  - repeating characters in, 80–82
- PLAY, 20, 157
- Play command, 130
  - PAL, 130, 147
  - Scripts, 147

- Pop (Ctrl-P), 160
- Pop command (Debugger), 160
- pop-up menus, 223
  - building, 224–228
  - See also* menus
- positive numbers, 60
  - formatting, 88
- POSTRECORD, 387–388
  - LEAVELOCKED keyword and, 388
- precedence, order of operator, 50
- prevent full locks, 381, 382
- prevent write locks, 380, 381
- primary indexes, 359
- private directories, 377
  - changing, 378
  - returning name of current, 377
- private variables, 42, 335
  - accessing, 336
  - debugging, 160
  - declaring, 100
  - dynamic scoping of, 100, 103, 335
  - miniscripts and, 159
- PRIVDIR(), 377
- PRIVTABLES, 377
- PROC, 109
  - CLOSED keyword and, 106
  - ENDPROC keyword and, 93
  - miniscripts and, 133
  - PRIVATE keyword and, 102, 335
  - USEVARS keyword and, 105, 335
- procedure data types, 32, 33
- procedure libraries
  - See* libraries
- procedure swapping, 354
  - eliminating, 356
  - See also* swap devices
- procedures, 91–114
  - autoloading, 109
  - branching commands and, 65
  - branching within, 68
  - calling, 94, 108–111, 354–355
  - changing, 113
  - closed, 105–107
  - debugging, 114, 153–154, 157, 160
  - defining, 92–99
  - DialogProc, 212, 267
  - See also* DialogProc procedure; WaitProc procedure
  - entering arguments in, 94–97
  - error, 358
  - error-handling, 121, 123–125
  - incorporating into applications, 370, 371
  - large, 354
  - losing, 113
  - memory and, 92, 106, 110, 354–355
  - naming, 38, 109, 113
  - nesting, 160
  - passing values to, 96
  - private variables and, 42, 100, 336
  - reading, 112
  - releasing from memory, 105, 107
  - return values in, 97–99
  - returning control to, 75
  - storing, 92, 378, 107–114
  - syntax, 92
  - terminating, 97
  - undefined, 109
  - updating, 109
  - variables in 99–104
  - WaitProc, 211, 215
- program
  - See* scripts
- programming, 6, 315
  - event-driven, 205–221, 237
- programming commands and functions, 21
  - See also* specific command or function
- PROMPT, 287
- prompts
  - responding to, 22, 144
  - testing, 370
- PROTECT, 116
- Protect command, 115, 116
- Protect command (Tools | More), 378
- protection
  - See* passwords
- prototypes, 120, 127, 130, 139, 371
- pull-down menus, 215, 223
  - building, 229–236
  - deactivating, 230, 236
  - nesting, 236
- push buttons, 243–244
  - creating, 243
  - default, specifying, 244
  - naming, 244

## Q

- queries, 330, 343, 367
  - executing with scripts, 26–27, 142
  - facilitating, 359
  - indexes and, 359, 360
  - locating specific records with, 325, 332
  - multi-table forms and, 309
  - multiuser applications and, 394

- problems with executing, 26
- referencing fields in, 50
- running, 349
- wildcard operators and, 57
- QUERY, 26
  - ENDQUERY keyword and, 26
- Query forms, 322
  - reproducing, 27
- query images, 22
  - See also* query statements
- query statements, 27
  - editing, 142
  - recording, 26, 141–143
  - saving, 367
  - tilde variables in, 41, 343–345
  - wildcard operators and, 344
- QuerySave command (Scripts), 26, 142, 367
- QuerySpeed command (Tools), 359
- queued events, 207
- QUIT, 19, 20, 76
- Quit (Ctrl-Q), 160
- Quit command (Debugger), 160
- QUITLOOP, 74

## R

- radio buttons, 253–255
  - adding titles to, 254
  - creating, 254
- read-only directories, 380–381
- read-only files, 377
- READLIB, 109, 110
  - IMMEDIATE keyword and, 110
- RECORD keyword, 285
- record locks, 381, 385–392
  - group, 391
  - multi-table forms and, 391–392
  - multiple, 390
  - releasing, 391
  - returning information on, 387
  - write-record, 392
- records
  - adding to tables, 338
  - assigning values, 327
  - changing, 73, 385, 387, 393–395
  - coediting, 387, 388
  - copying, 327
  - editing, 45
  - locating specific, 53, 325, 390
  - moving, 45
  - posting, 387, 388
  - reordering in tables, 350
  - restricting access to, 385
  - returning summary information on, 332
  - searching, 42, 359
  - storing, 45, 337
  - undoing changes to, 389
  - unlocking, 42, 385
  - updating, 385
  - working with, 326–327
- records, finding
  - See* LOCATE; queries; ZOOM
- records, locking
  - See* record locks
- RECORDSTATUS(), 387
- referential integrity, maintaining, 312–314
- REFRESH, 385
- Refresh (Alt-R), 385
- REFRESHCONTROL, 275–277
- REFRESHDIALOG, 277–279
- RELEASE PROCS, 105
- RELEASE VARS, 41, 44, 105
- REPAINTDIALOG, 268, 272
- RepeatPlay command
  - PAL, 130, 147
  - Scripts, 147
- reports, 367
  - calculated fields and, 37
  - multiuser applications and, 394
- reserved words
  - See* keywords
- RESET, 117, 327, 383
- restricted views, 307–310
- restructured tables, indexes on, 360
- RESYNCCONTROL, 269
- RESYNCDIALOG, 272
- retry period, automatic, 393
- RETRYPERIOD(), 393
- RETURN, 75, 97, 136
  - Retval and, 42
- Retval system variable, 42
  - assigning values to, 42
  - changing, 99
  - locks and, 382, 387, 393
  - procedures and, 98
- RMEMLEFT(), 358
- ROW(), 192
- RUN, 125
  - NOSHELL keyword and, 125
- Runtime
  - See* Paradox Runtime
- runtime errors, 121, 152
  - debugging, 158, 159

returning, 124  
trapping for, 123–125

## S

Sample application, 397–414  
  running, 398

sample tables, 162

SAVETABLES, 117

SAVEVARS, 41

Savevars script, 41, 119

.SC2 files, 353

SCAN, 73–74

scientific notation, 36

scope, 100, 103, 335, 336

  closed procedures and, 105  
  restricting, 335

screen attributes

*See* video attributes

screens

  echoing output to, 186–188

  repainting, 268

  updating, 394–395

Script Break menu, 132, 148

  miniscripts and, 136

Script Editor

*See* Editor

script elements, 143

  naming, 38

script errors, 386

  debugging, 159

script line, Debugger, 154

scripts, 15–30, 240

  accessing, from shared environments, 377

  choosing menu commands and, 22, 144

  combining, 141, 145, 147

  continuous play, 130, 158

  controlling sequence of execution in, 63

  creating, 120, 16–30, 139–146

  debugging, 41, 114, 134, 151–167

  editing, 115, 144–145, 161, 351

  entering strings in, 26, 144

  error-handling procedures and, 123, 124, 125

  executing queries from, 26–27, 142

  exiting to DOS from, 19

  including with applications, 372

  incorporating into applications, 369

  interrupting, 115, 149

  keyboard macros and, 347, 349

  keycodes in, 23, 144

  memory management and, 353–354

  mouse events and, 140

  naming, 372

  nesting, 16, 20, 160

  output, displaying, 135

  password protecting, 115–119, 378

  play, abandoning, 160

  playing, 19, 120, 130, 146–149

  predefined, 119

  problems with, 120, 148

  procedures in, 91, 92, 101

  recorded sequences and, 322

  recording, 130, 140–144

  renaming, 120

  restricting access to, 119

  retrieving, 145

  returning control to, 75

  sequence of execution, 20

  single-line, 133

  stepping through, 157

  streamlining, 93

  terminating, 19, 75, 76

  testing, 131, 134, 321

  variables and, 38, 335

  version compatibility, 8

  whitespace in, 16, 18

scripts, mini

*See* miniscripts

scroll bars, 259

secondary indexes, 134, 359

  building, 359

  maintaining, 360

  updating, 360

  using, 361

SELECT, 23, 324

SELECTCONTROL, 274

separator bars

*See* menu bars

separators, in arguments, 37

sequence of execution, commands, 19–20

sequential numbers, 330

set match characters, 80

SETCANVAS, 188

  image windows and, 186, 188

SETCOLORS, 200

SETKEY, 133, 134, 347–348, 351, 352

SETMARGIN, 193

SETPRIVDIR, 377

SETRECORDPOSITION, 311

SETRETRYPERIOD, 393

shared environments

*See* networks

shelling out to DOS, 125

short number (S) data types

- returning, 36
- SHOWARRAY, 238, 240–241
- SHOWDIALOG, 207, 212, 238, 241–267
  - ACCEPT keyword and, 258–259
  - CanvasElements parameter and, 242, 262
  - CHECKBOXES keyword and, 256–257
  - ControlElements parameter and, 242
  - EventList parameter and, 209, 242, 263
  - HEIGHT keyword and, 242
  - LABEL keyword and, 261
  - PAINTCANVAS keyword and, 262
  - PICKARRAY keyword and, 248–250
  - PICKDYNARRAY keyword and, 250–251
  - PICKDYNARRAYINDEX keyword and, 252–252
  - PICKFILE keyword and, 245–246
  - PICKTABLE keyword and, 247
  - PUSHBUTTON keyword and, 243
  - RADIOBUTTONS keyword and, 254–255
  - SLIDER keyword and, 259–261
  - TitleExpression parameter and, 242
  - triggers, 218, 221, 264, 267, 270
  - WIDTH keyword and, 242
- SHOWFILES, 238–240
- SHOWMENU, 224–226
  - control structures and, 66, 68, 72
  - DEFAULT keyword and, 224
  - TO keyword and, 224
  - UNTIL keyword and, 224, 225
- SHOWPOPUP, 226–228
  - CENTERED keyword and, 228
  - control structures and, 68
  - TO keyword and, 228
  - UNTIL keyword and, 224, 228
- SHOWPULLDOWN, 224, 229–236
  - control structures and, 68
  - DISABLE keyword and, 235
  - event processing and, 206, 215–216, 230
  - UNTIL keyword and, 224, 229, 230, 236
- SHOWTABLES, 238–240
  - TO keyword and, 239
- sign specifications, 88
- single-line scripts
  - See* miniscripts
- SLEEP, 135, 136
  - tracing through, 166
- slider controls, 259–261
  - creating, 260
  - incrementing, 260, 264
  - synchronizing, 264, 269
- Social Security numbers, 77
- sort order, 49
- statement sequence, 19
- statements, 15
  - branching, 69
  - See also* scripts
- status commands, 327
- status functions, 327
  - See also* specific status function
- status line, Debugger, 154
- Step (Ctrl-S), 157
- Step command (Debugger), 157
- streams, event, 207–209, 214
- string concatenation, 54–55
- string constants
  - See* alphanumeric constants
- string functions, 54
  - See also* specific string function
- strings, 32, 34, 342
  - ASCII characters in, 35, 58
  - assigning to fields, 52
  - assigning to variables, 40
  - backslash sequences in, 35, 58
  - case specifications and, 87
  - character, 58
  - combining, 132, 133, 156, 159
  - comparing, 54
  - concatenating, 47, 54–55
  - converting to dates and numbers, 55
  - default alignment, 87
  - editing, 132, 133, 156, 159
  - empty, 38
  - entering in scripts, 26, 34, 144
  - performing arithmetic operations on, 55
  - picture, 77–84
  - query statements and, 344
  - redisplaying, 132, 133, 156, 159
  - text, 26
- structures, control
  - See* control structures
- STRVAL(), 55
- STYLE, 194
- submenu
  - See* SHOWPOPUP; SHOWPULLDOWN
- submenus, building, 228
- submodes, 319
  - See also* system modes
- subroutines, creating, 20, 93
- subscript expressions, 44
- subscripts, 43, 426
  - See also* fixed arrays
- subtraction operator, 47
- summary information, 332
- swap devices, 355

*See also* memory  
 SWITCH, 64, 65, 66–68  
   building menus with, 66, 72, 225, 226  
   CASE keyword and, 66, 226  
   OTHERWISE keyword and, 66  
 SYNCCURSOR, 192  
 syntax, 15–18  
   control structures, 63  
   documentation conventions, 8  
   procedures, 92  
   testing, 132  
 syntax errors, 120, 152  
   debugging, 158  
 SYSCOLOR(), 200  
 SYSMODE(), 28, 319  
 system attributes, 195–201  
   overriding, 201  
 system modes, 317–320  
   abbreviated menu commands and, 28  
   changing, 318  
   minor, 319  
   returning current, 319  
 system time, displaying, 268  
 system variables, 42  
   assigning values to, 42  
 system-generated messages, 125

## T

tab characters, 35  
 TABLE keyword, 285  
 table locks, 379–384  
   multi-table forms and, 391  
   multiple, 384  
   placing, 380, 382, 384  
   releasing, 383  
   returning information on, 384  
 table names, 322  
   using variables as, 321  
 tables, 238, 245, 325, 396  
   checking for existence of, 321  
   coediting, 382  
   copying, 322  
   creating, 321, 322, 324  
   cross-tabulating, 394  
   declaring private, 377  
   editing, 382  
   indexing, 359–361, 367  
   interacting with, 285–286  
   large, 360  
   locking, 42, 379, 382, 384, 391  
   losing, 321

  moving through, 73, 325  
   overwriting, 321  
   password protecting, 115–119  
   referencing fields in, 50  
   renaming, 322  
   restricting access to, 119, 382, 384  
   restructuring, 360, 367  
   sample, 162  
   sorting, 133  
   temporary, 377  
   unlocking, 42  
   working with, 73, 96  
 tags  
   event, 45, 216–218  
   *See also* dynamic arrays  
 telephone numbers, 82  
 temporary tables, multiuser applications and, 377  
 terminating commands, 75–76  
   *See also* control structures  
 text, 193  
   editors, 16, 144, 145  
   editors, resident, 351  
   writing to screen, 193  
 text strings  
   entering in scripts, 26  
   *See also* strings  
 tilde (~) labels, in scripts, 244, 261  
 tilde (~) variables, in queries, 41, 343  
 TIME(), 268  
 time, displaying system, 268  
 Trace (Ctrl-T), 158  
 Trace command (Debugger), 158  
 transaction logs, 376  
 trigger cycle, 221  
 triggers, 218–221, 264, 267, 270, 289, 290  
   categories, 220  
   cycles, 290  
   processing, 221  
   trapping, 218  
   *See also* events  
 truncated fields, 87  
 tuple  
   *See* record  
 tutorials, Debugger, 162–167  
 TYPE(), 38, 44, 336  
 type-in boxes, 258–259  
   creating, 258  
   synchronizing, 264, 269, 272  
 typeahead buffer, 213  
 typecasting, 336–337  
 TYPEIN, 324

## U

- unary negation, 47
- unassigned variables, 132, 159
- UNDO, 389
- UNLOCK, 382, 383
  - Retval and, 42
- UNLOCKRECORD, 385–387, 388, 391
  - Retval and, 42
- UNPASSWORD, 116, 117
- uppercase characters, assigning to strings, 87
- user documentation, 374
- user interactions, 205, 211, 218, 220
  - denying, 220
  - dialog boxes and, 237, 238, 253, 255, 264
  - intercepting, 342
  - message events and, 208
- user interfaces, 169

## V

- ValCheck command, 78
- valid events, 216
- validity checks, 286
- Value command
  - Debugger, 156
  - PAL, 33, 131, 136
- Value script, 119, 132, 157
- values, 31
  - assigning to arrays, 43–46
  - assigning to fields, 50–53, 326, 333–334
  - assigning to procedures, 96
  - assigning to variables, 40, 336
  - calculating, 330–332
  - comparing, 48–49
  - control structures and, 67, 71, 75
  - decimal, 86
  - default, entering, 77
  - duplicate, indexes and, 361
  - fixed, 34
  - formatting, 77–89, 193
  - incorrect, 385, 395
  - locating specific, 325
  - losing, 40, 313
  - manipulating, 46, 51, 54–61, 333–345
  - negative, 88
  - optional, 81
  - positive, 88
  - repeating, in fields, 80
  - repetitive, 77
  - returning, 42, 53, 97–99
  - rounding, 87
  - saving, 41

- storing temporarily, 38, 43, 334
- testing, 40, 68, 370
- transferring, 338
- values, blank
  - See blank values
- values, logical
  - See logical values
- variables, 38, 342
  - arrays and, 43
  - assigning values to, 40, 336
  - changing, 153
  - closed procedures and, 105, 335
  - debugging, 157
  - declaring private, 42, 100, 336
  - dynamic scoping of, 100, 103, 335, 336
  - entering in expressions, 38–43, 334–337
  - global, 41, 99, 105
    - See also global variables
  - losing, 42, 105
  - naming, 38, 335, 337
  - passing in procedures, 100, 101
  - private, 335
    - See also private variables
  - procedures and, 99–104
  - query statements and, 41, 343
  - releasing, 41, 42, 105
  - resetting value of, 41
  - restoring, 41
  - saving, 41
  - table names and, 321
  - testing, 132
  - typecasting, 336–337
  - unassigned, 132, 159
- variables, system
  - See system variables
- video attributes, 194
  - color, 194, 195, 201
- VIEW, 382
- VROOMM, 356–358
  - See also memory

## W

- WAIT, 207, 211–212, 285–302
  - building menus with, 215, 233, 300
  - EventList parameter and, 209
  - FIELD keyword and, 212
  - multitable forms and, 296
  - pictures in, 77
  - RECORD keyword and, 212
  - Retval and, 42
  - TABLE keyword and, 212



- triggers, 218, 221, 289
- WORKSPACE keyword and, 212
- WaitProc procedure, 211, 215
  - arguments for, 289
- Where? (Ctrl-W), 160
- Where? command (Debugger), 160
- WHILE, 72–73
- whitespace, in scripts, 16, 18
- width specifications, 86
- wildcard operators, 57
  - query statements and, 344
- wildcards, DOS, 239, 245
- window attributes, 174–177, 188, 191
  - array index and, 175
  - returning, 174, 202
  - setting, 176, 177, 201, 204
- WINDOW CLOSE, 179
- window coordinates, specifying, 172
- WINDOW CREATE, 172, 189
  - ATTRIBUTES keyword and, 172, 176
  - FLOATING keyword and, 172
- WINDOW ECHO, 187
- WINDOW GETATTRIBUTES, 174
- WINDOW GETCOLORS, 201
- WINDOW HANDLE, 173
  - CURRENT keyword and, 173
- window handles 172–174
  - current canvas, 190
  - reassigning, 174
  - returning, 173
- WINDOW LIST, 173
- WINDOW MAXIMIZE, 177
- WINDOW MOVE, 177
- WINDOW RESIZE, 177
- WINDOW SCROLL, 177
  - SCROLLCOL keyword and, 177
  - SCROLLROW keyword and, 177
- WINDOW SELECT, 178
- WINDOW SETATTRIBUTES, 176
- WINDOW SETCOLORS, 204
- WINDOW(), 125
- windows, 171–184, 186
  - active, 181
  - arranging onscreen, 179–182
  - closing, 174, 179, 190
  - creating, 172, 174, 176, 181
  - current, specifying, 178, 181
  - drawing, 195
  - moving, 177
  - removing all, from workspace, 328
  - resizing, 177
  - scrolling, 177

- WINNEXT, 181
- workspace
  - clearing, 383
  - cursor, synchronizing with canvas cursor, 192
  - multi-table forms and, 310–312
  - restoring, 327
  - status commands and functions, 327–330
  - working with, 332
- write locks, 379–381
- write-record locks, 392
- WRITELIB, 109, 113

## Z

- z-order
  - objects and, 179
  - windows and, 181
  - See also* windows
- zeros
  - dividing numbers by, 48
  - multiplying numbers by, 48
  - returning in fields, 38
- ZOOM, 325, 359
- ZOOMNEXT, 325

# **Borland**

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 431-1000. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # PDX1045WW21775 • BOR 5929