

Formulas & Programming

Panorama Formulas & Programming (Version 6.0)
Copyright © 2010, ProVUE Development,
All Rights Reserved

ProVUE Development
18685-A Main Street PMB 356
Huntington Beach, CA 92648
USA

www.provue.com



Table of Contents



— Click on any entry to jump to the page —

Formulas	19
Formulas In Action	19
Displaying/Printing A Formula	20
Storing Formula Results in the Database	22
Using a Formula to Locate/Select Information.....	25
Formulas in Procedures.....	28
Using the Formula Wizard	29
Calculations with Database Fields	30
Changing the Active Database	32
Using Fields from Other Databases	33
Topic and Functions Help Menus	34
Function Search	36
Special Formula Result Formats	38
The Programming Reference Wizard	41
Formula Components	42
Formula Grammar	42
Calculation Order and Parentheses	43
Functions	43
Multi-Parameter Functions	44
Zero Parameter Functions.....	44
Functions Menu	45
Whitespace.....	46
Grammar Errors	47
Comments	49
Values	49
Constants.....	49
Build in Constants: Pi, Carriage Return and Tab	50
“Pipe” Delimited Constants.....	50
Fields	50
Using the Current Field	52
Line Item Fields	52
Variables	53
Variable Names	54

What's Inside A Variable?	54
The Life Cycle of a Variable	54
Creating Variables in a Procedure	55
Initializing Variables	56
Variables and Data Types	56
SuperObject Variables	56
Variable Name Conflicts	56
Permanent Variable Tips	57
Special Characters	57
Working With Extremely Complex Formulas	59
How Large Should the Buffer Be?	59
Arithmetic Formulas	60
Dividing by Zero	61
Overflow/Underflow Problems	61
Adding Line Item Fields	62
Basic Numeric Functions	62
Scientific Functions	64
Financial Functions	66
Text Formulas	67
Gluing Strings Together	67
Functions for Taking Strings Apart	68
Taking Strings Apart (Text Funnels)	69
Numeric Start and End Positions	70
Specifying Numeric Length Instead of Position	70
Start/End Positions by Character Matching	71
Cascading Text Funnels	71
Character Matching in Reverse Gear	72
Stripping Out Individual Words	73
Multiple Matching Characters for Start/End Position	74
Non-Matching Character for Start/End Position	75
Limitations of Text Funnels	77
String Testing Functions	78
String Modification Functions	80
Converting Between Numbers and Strings	84
Characters and ASCII Values	87
Working with Character Values	87
Invisible Characters	88
The ASCII Chart Wizard	89
Showing Character Ranges with the ASCII Wizard	91
ASCII Character Constant Functions	92
Text Arrays	93
Picking a Separator Character	93
Working With Arrays	94
HTML Tag and Tag Parsing Functions	101
Tag Parameter Functions	103
HTML Table Parsing Functions	104
HTML/URL Conversion Functions	104
HTML Generating Functions	105
Encoding/Decoding Base64 Data	105
Date Arithmetic	106
Today's Date	106
Converting Between Dates and Text	107
Date Functions	108

Calendar Functions	110
Time Arithmetic	113
Converting Between Times and Text.....	113
Time Calculations	114
Time Calculations with Text	116
Calculating Time Intervals Smaller Than One Second	116
Time Code Calculations (Video/Film)	117
SuperDates (combined date and time)	118
Reminders.....	119
Appointments vs. To-Do's	120
Creating and Modifying a Reminder.....	120
Reminder Functions	122
Alarms	123
True/False Formulas	124
Comparison Operators	124
A beginswith B.....	125
A endswith B	125
A contains B	125
A notcontains B	125
A soundslike B.....	125
A match B.....	126
A matchexact B	127
A notmatch B	127
A notmatchexact B	127
A like B	127
Combining Comparisons	127
A and B.....	127
A or B	128
A xor B.....	128
not A.....	128
Equals Comparison vs. Assignment.....	128
True/False Values.....	129
The ? Function.....	130
Converting a Boolean Value to Text.....	130
Linking With Another Database	131
The Lookup Wizard.....	132
Type Mismatch Problems	135
Lookup Variations.....	136
Looking Up Rates in a Rate Table.....	137
Looking Up Multiple Fields From One Record.....	137
The GrabData Function	139
Looking Up Multiple Values at Once.....	139
Linking Clairvoyance to the Lookup Key Field.....	142
Looking Up Data in the Current File	144
The Assign Function	144
Zip Code Lookup.....	145
US Post Office Abbreviation Functions	146
Graphic Co-Ordinates	146
Points.....	147
Rectangles.....	149
Colors.....	154
Raw Binary Data	156
The RPN Programmer's Calculator	161

Converting Between Different Bases	161
Calculations with Reverse Polish Notation	162
Boolean Operators	164
Disk Files and Folders	165
Resource Files	170
Import/Export Functions	174
System and Database Information Functions	175
System Information	175
User Information	179
Variable Information	180
Database Information	180
Window, Form and Report Information	187
Server Database Information (Panorama Enterprise)	194
Custom Functions	196
The Custom Functions Wizard	196
Function Names	197
Parameter Names	198
Advanced Topic: The FDF File	198
Advanced Topic : Creating Custom Functions In A Procedure	198
The Custom Functions (ProVUE) Wizard	199
Procedures	201
Programming Isn't Magic!	201
Introduction to (Panorama) Programming	201
Procedures	202
Statements	202
A Simple Procedure in Action	203
Creating a Procedure with the Recorder	210
Recording Mouse Clicks	212
Non Recordable Menus and Tools	213
Recording Data Entry	213
Writing a Procedure from Scratch	214
Writing Statements	215
Trying Out a Procedure	216
Checking for Mistakes	218
Mysterious Errors	220
Closing the Window When a Procedure is Finished	220
Re-Opening a Procedure	220
Font and Size	221
Adding a Recording to an Existing Procedure	221
Programming Helpers	223
The Programming Assistant Dialog	223
Using the Assistant from the Keyboard	224
Assistance Domains	225
Getting Assistance with a Selection	227
Smart Text Insertion	227
The Programming Context Menu	228
Help Submenu	228
Mark Submenu	230
Insert Field Name Submenu	231
Insert Form Name Submenu	231
Insert Procedure Name Submenu	231
Topics Submenu	232

Opening a Procedure or Form.....	233
Selecting Parentheses Contents	233
Comment/Uncomment	234
Programming Reference Wizard	235
Navigation Using the Search Panel and Topic List	235
The Full Text Search Option	237
Navigation Using the Topic, Statement and Function Menus	238
Navigation Using HyperLinks	238
Built In vs. Custom Statements and Functions.....	239
Using the Template Panel	239
Minimizing the Programming Reference Wizard	240
Data Flow	241
Assignment Statements.....	241
Triggering Automatic Calculations.....	241
The Define Statement	242
The Set Statement	242
The FormulaValue Statement	243
Variables.....	245
Creating a Variable.....	245
Assigning a Value to a Variable	246
Using a Variable in a Formula	246
The Birth and Death of a Local Variable	247
Long Life Variables.....	247
Destroying a Variable	247
Variable Accessibility.....	248
Accessing “Dormant” Variables	248
“Hidden” Variables and Fields	249
Accessing Variables In Form Objects (Text or Images)	249
Creating Variables with a SuperObject	249
Permanent Variable Tips.....	251
Displaying and Changing Variables	251
Control Flow	253
True/False Formulas.....	253
Equals Comparison vs. Assignment.....	254
True/False Values	254
IF Statements	255
ELSE Statements	255
Nested if Statements	255
Error Handling with if error	256
CASE Statements.....	257
LOOP Statements.....	257
Stopping a Loop in the Middle	258
Restarting a Loop in the Middle.....	259
Subroutines.....	259
CALL Statement	259
Calling Procedures With Unusual Names	260
Passing Values to a Subroutine (Parameters)	261
Passing Values Back From a Procedure.....	262
What if the parameters don’t match the procedure?	265
Calling a Subroutine in Another Database	267
Terminating a Subroutine in the Middle.....	267
Mini Subroutines within a Procedure	268
Subroutines and Local Variables.....	269

The UseCallersLocalVariables and UseMyLocalVariables Statements	270
Recursive Subroutines	271
Using a Subroutine in a Formula (the CALL(function).....	271
Restrictions on Subroutines used as Formulas.....	275
Other Control Flow Statements	275
Jumping to an Another Location in the Program	275
Stopping the Program	276
Aborting a Program	276
Controlling the Abort Process.....	277
Doing Nothing for a While	278
Building Subroutines On The Fly (The Execute Statement)	278
Tips for On-The-Fly Program Writing	280
Execute and Local Variables.....	282
Using Execute to Process Arrays.....	282
Do It Yourself Data Merge.....	284
On-The-Fly Subroutine Error Checking.....	285
Building Parameters on the Fly (Parameters in a Variable)	286
Catching Program Errors (Especially for Web and other Server Applications).....	286
Custom Statements	287
The Custom Statements Wizard	289
Creating Your Own Custom Statement Library	291
Creating a New Custom Statement.....	291
Setting Up a Procedure Information Block	294
Processing Parameters	296
Optional Parameters	296
Repeating Parameters	297
Raw Parameters.....	297
Debugging a Custom Statement	298
Accessing Forms & Procedures in the Library Database.....	298
Advanced Topic: Using Libraries In Other Folders.....	298
Program Formatting.....	299
Notes To Yourself.....	302
“Commenting Out” Statements.....	302
Organizing Large Procedures (The Mark Menu)	303
Suppressing Display of Text and Graphics.....	305
Updating the Display After (or Within) a NoShow Block.....	305
ShowPage	306
ShowLine.....	306
ShowFields field,field,...,field	306
ShowColumns field,field,...,field.....	307
ShowVariables var,var,...,var	307
ShowRecordCounter	307
ShowOther field,code	308
Checking NoShow Status.....	308
Disabling the Watch Cursor	308
Hide and Show	309
Debugging a Procedure.....	310
The Panorama Interactive Debugger.....	313
The Debug Statement	313
Using the Debugger	313
Single Stepping	314
Resuming Full Speed Execution	316
Making Corrections to a Procedure.....	316

Watching Computations	316
Using the Inspector to Examine Fields, Variables and Formulas	318
What Fields or Variables can be Displayed?	321
Displaying Functions	322
Error Detail Wizard	323
Using the Error Detail Wizard	324
Finding the Source of the Error	325
Open Reference Wizard	327
Copy to Clipboard	327
Error Detail Problems	327
Debugging with the TTY (Virtual Teletype) Wizard	328
Using TTY with Growl	331
Selective TTY Output (Modes)	332
Keeping a Permanent Record	333
Procedure Debug Log	334
The Procedure Log Window	334
Recording a New Log	335
Decoding Parameters and Assignment Statements	339
The LogMessage Statement	340
The Log Menu	341
Using the View Wizard with Procedures	342
Searching All Procedures	342
Displaying Source Code Statistics	345
Exporting and Importing Procedure Source Code	346
Cross Referencing	347
The Cross Reference Wizard	347
Opening a Form, Procedure or Crosstab	349
Setting up a New Cross Reference	351
Updating a Cross Reference	352
50 Ways to Trigger a Procedure	353
The Action Menu	353
Action Menu Options	354
Setting Different Menu Item Styles (Bold, Italic, etc.)	354
Shortcuts/Command Key Equivalents	354
Disabled Menu Items	355
Separator Lines in a Menu	355
Renaming the Action Menu	357
Dividing the Action Menu into Multiple Menus	358
"Unlisted" Procedures	359
Live Menus	360
The FileMenuBar Statement	360
The .CustomMenu Procedure	361
Programming the .CustomMenu Procedure	362
The info("trigger") Function	362
Processing Custom Menus with Simple IF's	363
Processing Custom Menus with Nested IF's	364
Splitting the Trigger into Menu/Item Names	364
Menus with Modifier Keys	365
Building Menus from Arrays	365
Command Key Equivalents	366
Menu Styles	367
Disabled Menu Items and Separator Lines	367
Submenus (Hierarchical Menus)	368

Multiple Column Menus	369
The WindowMenuBar Statement.....	370
Advanced Topic: Live Menus Behind the Scenes	370
Menu Titles.....	370
Menu Items.....	371
Submenus	372
Formula Errors	372
The menu(, menuitems(, arraymenu(and checkarraymenu(functions.....	372
Helper Functions for Standard Menus.....	373
Buttons.....	374
Hidden Triggers	376
Creating Hidden Trigger Procedures.....	376
.About	377
AutoGrow.....	377
.ClearRecord.....	378
.CloseWindow.....	378
.CurrentRecord	379
.CustomMenu	379
.DeleteRecord.....	379
.DialogKeyDown	379
.Help	380
.Initialize.....	380
.KeyDown	380
.ModifyRecord.....	381
.ModifyFill.....	382
Logging Changes (Audit Trail) with .ModifyRecord, .ModifyFill and info("modifiedfield") ...	383
.NewRecord.....	384
.OutOfBounds.....	385
.ZoomFailed.....	385
Data Entry Triggers.....	385
Data Entry Triggers (Part Two).....	387
Hot Key Procedures.....	388
HotKeys with Modifiers.....	389
Universal HotKey Procedure	389
Triggering a Procedure Every Second.....	389
Triggering a Procedure Every Minute.....	391
Triggering a Procedure As Soon As Possible	391
Event Handler Procedures.....	392
Text Editor SuperObject ..Handler Option	393
Focus Procedure	393
..OpenForm	394
..ActivateForm	394
..CustomAbout.....	395
..CloseDatabase.....	395
Programming Techniques.....	397
Accessing Files.....	397
Files and Folders	397
Combined Folder Location and File Name	398
Folder ID's and Paths	399
Locating a File with Standard Dialogs	400
Customizing the Standard File Dialogs	403
Opening a Panorama Database	403

Suppressing the Default Extension	403
Appending Databases End-to-End	404
Eliminating Duplicates in Appended Data	404
Replacing the Data in a Database.....	404
Saving a Panorama Database.....	405
Closing a Database	405
Shutting Down Panorama.....	406
Importing Text Files	407
Carriage Returns in the Data.....	407
Importing a Text File into an Existing Database	407
Importing from a Variable	408
Importing HTML Tables	408
Re-Arranging the Order of Imported Data	408
Building the ImportUsing Formula on the Fly	410
Exporting Text Files	411
Exporting Line Items as Separate Records.....	412
Analyzing Line Items	413
Exporting Array Elements as Separate Records.....	413
Opening a Document in Another Application.....	414
Smart Merge Synchronization	415
How Smart Merge Synchronization Works.....	415
Adding Smart Merge to Your Database	415
The Modified Field.....	416
Adding New Records.....	416
The Smart Merge Procedure.....	417
Directly Reading and Writing Disk Files.....	419
What's in a File?	419
Reading Data Files.....	420
Reading Really Big Data Files.....	421
Writing Data Files	422
Copying Data Files	423
Using FileSave and ArrayBuild to Export Data.....	423
Reading and Writing Resource Forks	426
Erasing a File	427
Changing a File's Name	427
Changing a File's Type and Creator.....	427
Creating a New Folder	427
Getting Information about a File	428
Getting and Setting Additional File Information	429
Accessing and Modifying File Permissions	429
Building a List of Files or Folders	429
Building a List of Disks (Volumes).....	430
Working with Resources	431
Opening and Closing Resource Files.....	433
Opening a Resource File in the .Initialization Procedure	433
Reading a Resource.....	434
Reading STR and STR# Resources	434
Writing a Resource.....	435
Deleting a Resource	436
Renumbering a Resource	436
Listing Resources.....	436
Working with Multiple Resource Files	438
Accessing the Windows Registry.....	439

Getting Information About Registry Items	439
Modifying Registry Entries.....	440
Deleting a Registry Entry.....	441
Monitoring Memory Usage.....	442
Windows	443
Opening a Window	443
Specifying the New Window Location.....	444
New Window Options	446
Non Standard Window Styles.....	447
Changing a Window's Position/Options	448
Changing a Window's View	449
Changing the Name of a Window.....	449
Scrolling Inside a Form Window	449
Closing a Window.....	450
Trapping the Close Box.....	451
Changing The Window Order (Who's on Top?).....	451
Temporary "Invisible" Windows	452
Databases Without Windows	453
"Magic" Windows	454
Window Clones.....	455
Designing A Clone Window Application	456
Alerts.....	462
Displaying a Message.....	462
Alerts With Multiple Buttons.....	464
The Alert Statement	467
Obsolete Alert Statements	469
Suppressing Alerts	470
The SuperAlert Statement	470
The DisplayData Alert.....	477
Dialogs.....	478
Basic Text Entry Dialogs.....	478
The SuperGetText Statement.....	480
Obsolete Text Entry Statements.....	483
The SuperChoiceDialog Statement	483
Custom Dialogs	487
Preparing a Form for Use as a Dialog.....	487
Customizing the Dialog Code.....	493
Options to the RunDialog Statement.....	497
Editing Database Information with a Dialog	499
Custom Dialog Menus	501
Accessing and Modifying the Database Structure (Fields)	503
Getting Information About Field Structure	503
Modifying Field Structure Directly	504
Hiding and Showing Fields	506
Working With the Design Sheet.....	507
Updating Database Structure From Another Database.....	507
Transferring Permanent Variables	508
Verifying Database Identity	509
Database Navigation and Editing.....	510
Moving Up and Down in the Database	510
Moving Left and Right.....	513
Moving "Left" and "Right" on a Form	514
Moving to an Empty Line Item Field.....	516

Adding and Deleting Records	517
Modifying the Database One Cell at a Time	520
Accessing and Modifying the Current Cell	520
Accessing and Modifying the Clipboard	521
Triggering Automatic Calculations.....	521
Triggering Automatic Procedures	522
The Set Statement	522
The FormulaCalc Statement	523
Opening the Input Box.....	523
“Natural” Data Entry.....	525
Natural Data Display	526
Natural Data Entry	528
Validating a Credit Card Number.....	531
Moving Data Between Files	532
Cross Database Assignment Statements	532
Identifying Data to Move	532
Transfer Function Parameters	533
Single Record Transfer Functions	535
grabdata(target database,data field)	535
lookup(target database,key field,key value,data field,default value,summary level).....	535
lookuplast(target database,key field,key value,data field,default value,summary level)	535
lookupselected(target database,key field,key value,data field,default value,summary level)	536
table(target database,key field,key value,data field,default value,summary level).....	536
Clairvoyance and Lookups	538
The SpeedCopy Statement (Multiple Assignments in One Statement)	538
Multiple Record Transfer Functions.....	540
lookupall(target database,key field,key data,data field,separator)	540
lookupcalendar(target database,key field,key data,data field,separator)	540
lookupptime(target database,key field,key data,pattern,separator)	540
After a Lookup...Modifying the Original Data	541
Using Lookups for Display/Printing.....	542
Using ArrayBuild to Transfer Data Between Files	543
Posting Data to Other Databases	546
The Post Statement.....	546
The PostAdjust Statement.....	547
Sorting.....	549
Reducing Screen “Flashing”	549
Making Sorts Even Faster	549
Locating Information	550
Finding Information	550
A Handy Universal Find Procedure	551
Find Next	553
Selecting Information	555
Handling Empty Selections	556
Selecting Duplicates	557
Live Clairvoyance™.....	557
Adding a Cancel Search Button	563
Clicking on the Live Clairvoyance List Object	563
Summaries and Outlines.....	564
Summary/Outline Examples	565
Calculating Grand Totals	567
Running Total	569
Running Difference	569

Transforming Big Chunks of Data	570
Making Transformations Even Faster	571
Numeric Calculations with FormulaFill	571
Suppressing Zero's	572
Fill vs. FormulaFill	573
Using FormulaFill to Transform Text	575
Date Calculations with Formula Fill	577
The SEQ Function	578
Filling Empty Cells	579
Automatic Numbering	581
Propagate and UnPropagate	582
Using UnPropagate to Eliminate Duplicates	582
Change (Find and Replace)	582
Changing with the Replace(Function	585
Data Style and Color	586
Accessing Style and Color in a Formula	588
Processing/Transforming an Entire Array	589
"Filtering" an Array	589
Stripping Blank Elements From An Array	590
Reversing the Order of an Array	591
Using Regular Text Functions with Arrays	591
Sorting an Array	591
Removing Duplicate Items from an Array	592
Building an Array from a Database	592
Appending an Array to a Database	593
Copying Between Multiple Variables and an Array	593
Editing an Array using Separate Variables	595
Processing/Transforming Binary Data	596
Bits	596
Bytes	596
Words	597
Long Words	597
Creating Binary Values	597
One Dimensional Arrays of Binary Values	598
The CharacterFilter Statement	598
The ChunkFilter Statement	599
The TextFilter Statement	599
Data Dictionaries	600
Key/Value Pairs	601
Storing a Key/Value Pair in a Dictionary	601
Accessing Dictionary Entries	602
Another Technique For Initializing a Dictionary	603
Additional Methods for Modifying Dictionary Entries	603
Looking Up a Dictionary Key Given Its Value	604
Accessing the Internet	605
Basic Web Access	605
Fetching Web Pages	605
Parsing Web Pages	606
Fetching Images	611
Relative URLs	611
Submitting Forms	612
Cookies	614
Accessing Web Content	615

Generating Map Images	615
Generating the Same Map at Different Zoom Levels (Scales) or Sizes.....	616
Adding an Interactive Map Interface to a Database	617
General Zip Code Information	620
Street Address Information.....	621
White Pages	622
FedEx Shipment Tracking	623
Controlling Web and E-Mail Clients	624
Displaying a Web Page	624
Displaying a Web Page on a Local Hard Drive	625
Displaying a Map	625
Sending an E-Mail	626
Sending E-Mail	627
Channel Configuration.....	628
Sending a single e-mail	629
Sending multiple e-mails	629
Programming Graphic Objects on the Fly	631
Basics of Graphic Object Programming.....	631
Selecting an Object by Name	631
Selecting Multiple Objects	631
Getting Information About Individual Objects	632
Modifying Selected Objects	637
Getting Information About Selected Objects.....	640
Object ID Values.....	641
Redrawing an Object	641
Dragging a Rectangle.....	642
Movable Dividers	646
Drag and Drop	648
Drag Items and Flavors	648
The Dropalyzer Wizard.....	649
Dragging Items from Panorama.....	650
Dragging a Single Flavor.....	650
Dragging Multiple Flavors.....	651
Receiving Dragged Data.....	652
The .DropProcedure	652
Dropping Files and Folders on Panorama.....	654
VCard Drag and Drop.....	654
Drag and Drop (Obsolete Method)	656
Program Control of SuperObjects™	664
The Active SuperObject.....	665
Accessing and Modifying a SuperObject's Internal Data.....	667
Internal Data Types	667
Text Editor SuperObject Commands	668
Text Editor Internal Data	674
Text Display SuperObject Internal Data	675
Word Processor SuperObject Commands.....	676
Word Processor Internal Data	687
Super Flash Art Commands (Including Movie Control)	688
Super Flash Art Internal Data	691
Converting Between Image Formats	692
Working with JPEG Images	693
Taking an iSight Snapshot.....	694
Building Web Like HyperText Systems with Super Flash Art	695

Preparing Pictures with Extractable Text	695
Programming a HyperText Engine	697
Extracting All Text of a Specific Style	699
Creating Multi-Page Pictures.....	700
Push Button Internal Data	701
Flash Art Push Button Internal Data.....	702
Data Button SuperObject Internal Data.....	702
Flash Art Data Button SuperObject Internal Data	703
Sticky Push Button SuperObject Internal Data.....	704
Pop-Up Menu SuperObject Internal Data.....	705
List SuperObject™ Commands	706
Using Drag and Drop to Change the Order of Items in a List.....	710
List SuperObject Internal Data	711
Auto Grow SuperObject™ Commands (Elastic Forms).....	712
Auto Grow SuperObject Internal Data	712
Super Matrix SuperObject™ Commands	713
Super Matrix SuperObject Internal Data	715
Scroll Bar SuperObject™ Commands	716
Speech Synthesis	717
The Speak Statement.....	717
Embedded Speech Commands	717
The StopSpeaking Statement.....	717
The info("speaking") Function.....	717
Buffered Speech	718
Speaking Using the Speech Wizard	719
Printing.....	720
Selecting a View for Printing.....	720
Selecting a Printer	720
Changing the Current Printer	720
Changing the Default Printer	720
Getting Information About Printers	721
Adjusting Page Setup	721
Preparing Data For Printing	721
Printing the Database	721
Printing a Single Record	722
Print Preview.....	722
Printing Using an Alternate Form.....	723
Printing Data in an Array.....	724
Printing Directly to a PDF File.....	725
Installing the CUPS-PDF Package	726
Form Comments	730
The FormSelect Statement	732
Reading and Modifying Form Comments in a Procedure	733
Accessing and Modifying Procedures.....	734
Accessing a Procedure's Source Code	734
Changing a Procedure's Source Code	734
Creating a New Procedure	734
Storing Procedures in a Dictionary	735
Writing Your Own Channel Modules.....	737
The ModuleInformation Procedure	737
Channel Specific Procedures	738
The Channel Workshop Wizard.....	739
Previewing the ModuleInfo Procedure	741

Creating the Module	742
Working With Alternate Programming Languages.....	743
Choosing a Language.....	743
AppleScript	743
Shell Scripts.....	745
Scripting Languages	747
Perl	748
Ruby	749
Python	750
PHP	751
Code Embedding 101	752
Quoting Embedded Code	752
Embedding Code from a Text File	753
Using Panorama Fields and Variables as Terms in an Embedded Program	753
Using the Field Menu to Insert Fields Names	754
Using a Panorama Formula as a Term in an Embedded Program	754
Getting Data (Results) Back from Embedded Code	755
Standard Output (stdout)	755
Code Embedding Functions	755
Code Embedding Statements and the ScriptResult Variable.....	756
Bringing the Embedding Data Output Directly into a Panorama Field or Variable	757
Advanced Embedding Topics	758
The Embedded Code Pre Processor.....	758
Transferring Dates from Panorama to Embedded Code.....	759
Using Panorama Formulas “Bare” within an Embedded Program	759
Generating Constant Values	760
The External Script Wizard.....	761
Dealing With Errors in Embedded Programs	762
Working with External Editors	763
Working with External Debuggers	764
AppleScript Debuggers	764
Shell Scripts	765
Perl Debuggers	766
Ruby Debuggers	768
Special Embedding Options	769
Specifying the Maximum Embedded Program Runtime.....	769
Running Shell Scripts with Temporary Root Privileges (SUDO)	769
Specifying the Embedded Code Folder.....	770
Real World Embedded Code Examples	770
AppleScript — Address Book Search.....	771
Shell Script — File Info	774
Perl — POP3 Mail Reader.....	776
Ruby — Verify Email Domains	779
Python — HTML + Plain Text Email with Images.....	780
PHP — Extract EXIF Information from Images.....	785
Using AppleScript to Control Panorama from Other Applications.....	788
Everything You Really Need to Know... ..	788
Value of Cell	788
Executing Panorama Procedures.....	789
Transferring Data Between AppleScript and a Panorama Program.....	790
Transferring a Value Back From Panorama to the AppleScript (Returning a Value)	790

Working with Lists 791

AppleScript & Panorama... The Rest of the Story 792

 The Required Suite 792

 The Core Suite 793

The Objects 794

Chapter 1: Formulas



The result we proceed to divide, as you see,
by Nine Hundred and Ninety Two:
Then subtract seventeen, and the answer must be
Exactly and perfectly true.

- Lewis Carrol, The Hunting of the Snark

Panorama's primary job is storing and retrieving data. The primary job of formulas is to combine and manipulate data, both numeric and textual. Using formulas Panorama can automatically add up all the items in an invoice and calculate the sales tax. Using formulas Panorama can automatically divide all the names in a database into separate first and last names, or convert all the company names in a database to upper case. Using formulas Panorama can automatically look up the price of an item in inventory, or check the quantity on hand, or look up and display the items on a customer's previous order. As you can see, you'll need to learn how to use formulas to get the most from your Panorama investment. Fortunately, formulas are easy to learn and use (especially the most common mathematical formulas like totals, taxes and percentages). (However, we have to admit that sometimes formulas can be frustrating because they are very picky. If you get one little detail wrong, the formula won't work correctly. This isn't just a problem with Panorama, but with virtually any computer program that uses formulas. To help ease the potential frustration factor Panorama has some wizards you'll learn about later in this chapter that can help you build error free formulas.)

Formulas In Action

Formulas are a general purpose tool that Panorama can use in a variety of different situations. You can display or print the result of a formula, use a formula to modify the database, or use a formula to help locate information in the database. The next few sections demonstrate each of these techniques.

Displaying/Printing A Formula

A formula can be displayed or printed anywhere on a form with an auto-wrap text object (see “[Displaying Formulas in Auto-Wrap Text](#)” on page 602) or Text Display SuperObject (see “[Text Display SuperObjects™](#)” on page 608 of the *Panorama Handbook*). For example, consider the database shown below. The auto-wrap text object contains two formulas, one which calculates the total of the four columns (A, B, C and D) and one which calculates the average.

The screenshot shows two windows. The top window, titled "Numeric Formulas", displays a table with the following data:

City	A	B	C	D
Anaheim	3.25	7.31	14.82	1.93
Bakersfield	2.29	8.26	12.91	3.02
Camarillo	2.83	9.19	15.11	2.54
Fullerton	2.59			
Laguna Beach	3.06			
Newport Beach	3.18			
Whittier	3.67			

The bottom window, titled "Numeric Formulas:Just Testing (100%)", shows a text object with the formula: "For «City», the total is {A+B+C+D} and the average is {(A+B+C+D)/4}". Red circles and arrows highlight the formulas, with labels "formula to calculate total" and "formula to calculate average".

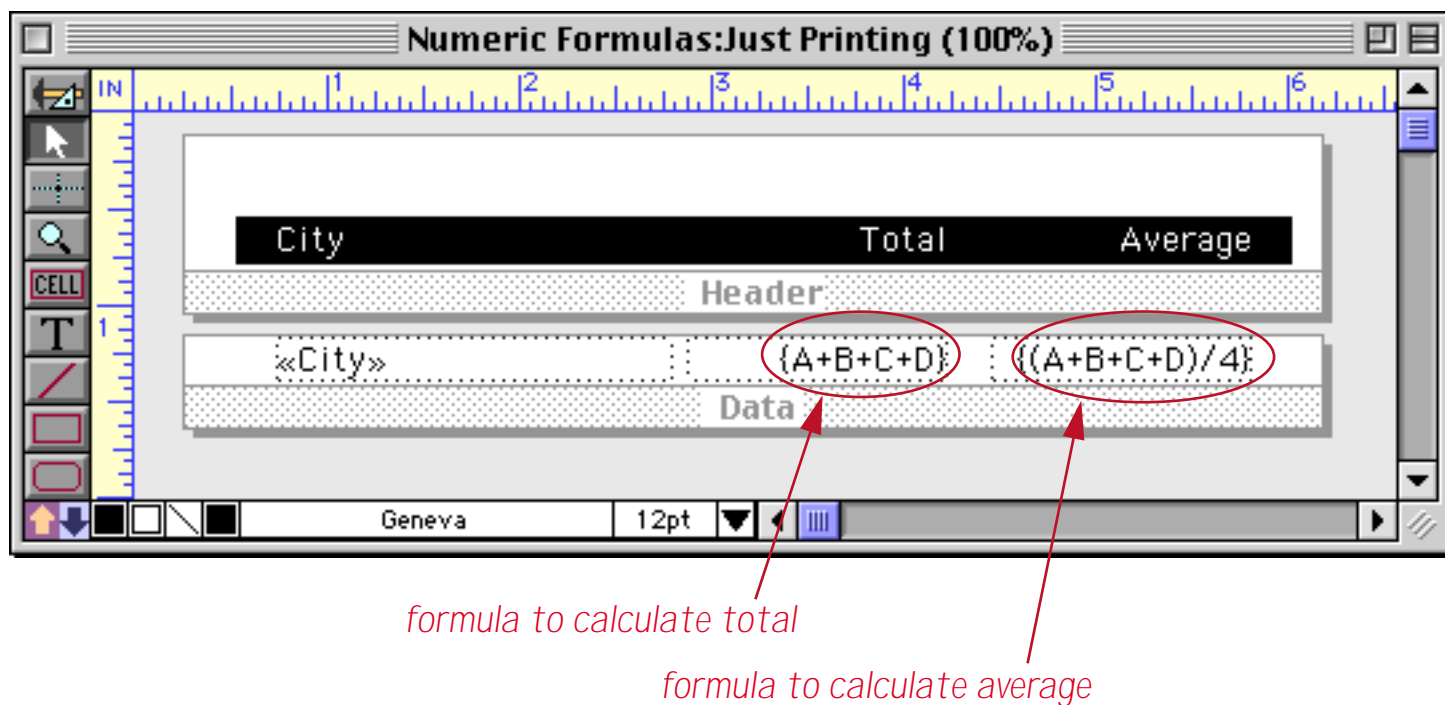
When the form is switched to Data Access Mode, Panorama calculates the formula results and displays them.

The screenshot shows the same two windows. The top window, titled "Numeric Formulas", displays the same table as before. The bottom window, titled "Numeric Formulas:Just Testing", now displays the calculated results: "For Anaheim, the total is 27.31 and the average is 6.8275".

When a formula is used this way the results are not stored anywhere in the database, they are simply calculated on the fly and displayed or printed, then thrown away. If you switch to a different record Panorama will calculate and display the new values.

The screenshot shows the same two windows. The top window, titled "Numeric Formulas", displays the same table as before. The bottom window, titled "Numeric Formulas:Just Testing", now displays the calculated results for Bakersfield: "For Bakersfield, the total is 26.48 and the average is 6.62".

You can even print a report using formulas calculated on the fly. (See “[Custom Reports](#)” on page 1061 of the *Panorama Handbook* to learn more about creating a custom report like this.)



Once again, the formula results are calculated on the fly as the report is printed, then discarded. Here is the finished report.

City	Total	Average
Anaheim	27.31	6.8275
Bakersfield	26.48	6.62
Camarillo	29.67	7.4175
Fullerton	26.09	6.5225
Laguna Beach	28.02	7.005
Newport Beach	25.1	6.275
Whittier	26.41	6.6025

(You may notice that the columns in the report above don't line up because they don't all have the same number of places after the decimal point. You can fix this with the `pattern()` function, see “[Converting Between Numbers and Strings](#)” on page 84).

Panorama's Flash Art feature allows a formula result to be displayed visually. See “[Flash Art™](#)” on page 750 of the *Panorama Handbook*.

Storing Formula Results in the Database

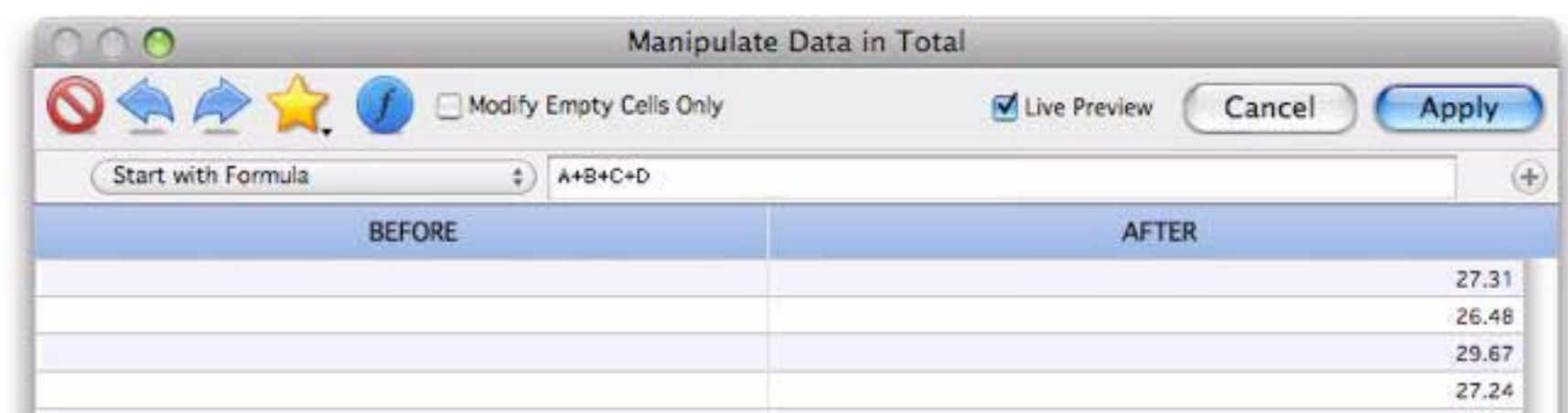
Sometimes you'll want to actually store the result of a formula in the database itself. You can do this manually after data has already been entered, or automatically as data is entered or changed. To illustrate these two techniques we'll add two new columns to the example database used in the last section, **Total** and **Avg**.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93		
Bakersfield	2.29	8.26	12.91	3.02		
Camarillo	2.83	9.19	15.11	2.54		
Fullerton	2.59	8.25	13.48	1.77		
Laguna Beach	3.06	9.45	12.5	3.01		
Newport Beach	3.18	7.22	12.32	2.38		
Whittier	3.67	5.23	14.24	3.27		

To calculate the values for these new fields we need to use the **Manipulate Data in Field** command (see “[The Manipulate Data Dialog](#)” on page 434 of the *Panorama Handbook*). To calculate the total, first click anywhere in the appropriate column.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93		
Bakersfield	2.29	8.26	12.91	3.02		
Camarillo	2.83	9.19	15.11	2.54		
Fullerton	2.59	8.25	13.48	1.77		
Laguna Beach	3.06	9.45	12.5	3.01		
Newport Beach	3.18	7.22	12.32	2.38		
Whittier	3.67	5.23	14.24	3.27		

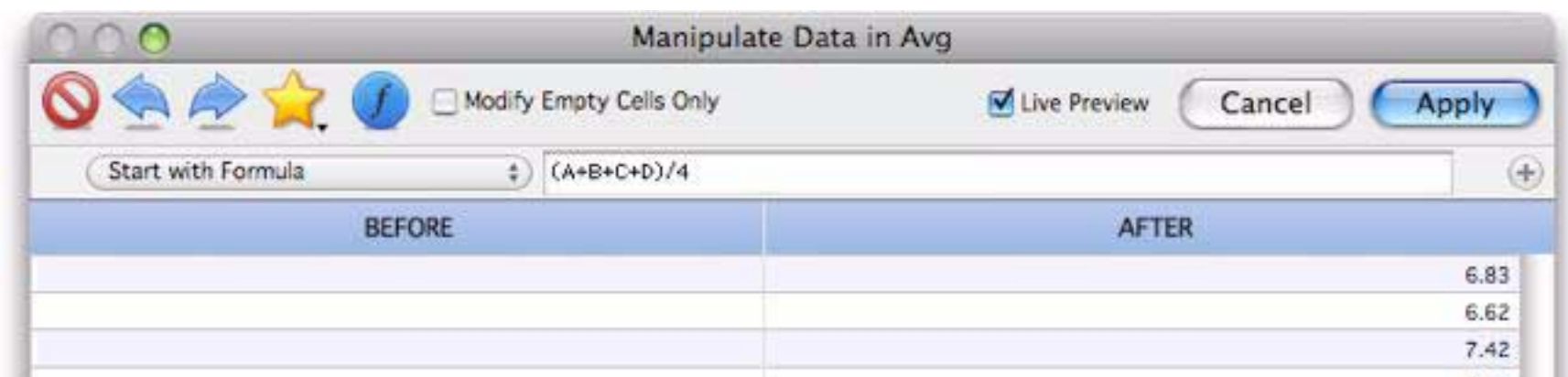
Now choose **Manipulate Data in Field** from the **Fields** menu, change the pop-up menu to **Start with Formula**, then enter the formula for calculating the total.



When you press **Apply** Panorama will calculate and store the value for every selected record in the database. In this case it performs seven calculations and stores seven values.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	
Bakersfield	2.29	8.26	12.91	3.02	26.48	
Camarillo	2.83	9.19	15.11	2.54	29.67	
Fullerton	2.59	8.25	13.48	1.77	26.09	
Laguna Beach	3.06	9.45	12.5	3.01	28.02	
Newport Beach	3.18	7.22	12.32	2.38	25.10	
Whittier	3.67	5.23	14.24	3.27	26.41	

Repeat the same process for the average, but of course with a different formula.



Here's the end result.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.10	6.27
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

Once the **Formula Fill** calculation is finished the formula is forgotten and the numbers are simply stored in the database just like a number that has been typed in. You can even manually edit a value to override the result of the formula calculation.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	25.7
Newport Beach	3.18	7.22	12.32	2.38	25.10	6.27
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

Since the **Formula Fill** formula is forgotten as soon as it is complete, Panorama does not update the values if the original numbers (in this case A, B, C or D) change or if new records are added to the database. If you want values stored in the database to update automatically as the database is updated you must set up automatic calculations in the design sheet (see "[Automatic Calculations](#)" on page 303 of the *Panorama Handbook*). Here's the design sheet for our example updated to automatically calculate the total and average.

Field Name	Type	Diç	Align	Out	Inp	Range	Choi	Link	Clair	Tab	Cap	Dup	Def	Equation	Reac	Writ	Wid	Notes
City	Text	0	Left			Any		Off	Off	Off	Yes				0	0	11	
A	Num	Fic	Right			Any		Off	Off	Off	Yes				0	0	4	
B	Num	Fic	Right			Any		Off	Off	Off	Yes				0	0	4	
C	Num	Fic	Right			Any		Off	Off	Off	Yes				0	0	4	
D	Num	Fic	Right			Any		Off	Off	Off	Yes				0	0	4	
Total	Num	Fic	Right	#,.	*	Any		Off	Off	Off	Yes			A+B+C+D	0	0	4	
Avg	Num	Fic	Right	#,.	*	Any		Off	Off	Off	Yes			(A+B+C+D)/4	0	0	4	

formula to calculate total
formula to calculate average

To activate these formulas you need to create a new generation for this database (see "[Database Generations](#)" on page 212 of the *Panorama Handbook*). Once you've done this you can start entering or updating information. In this illustration a new record has been added (**Diamond Bar**) and the first number typed in (but not entered into the database yet).

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13					
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.70	6.27
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

As soon as the data is entered by pressing the **Tab** (or **Enter**) keys the formulas update the **Total** and **Avg** fields.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13				3.13	0.78
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.70	6.27
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

formulas in design sheet update fields as data is entered

As more data is entered the **Total** and **Avg** fields are updated instantaneously.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13	7.81			10.94	2.73
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.70	6.27
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

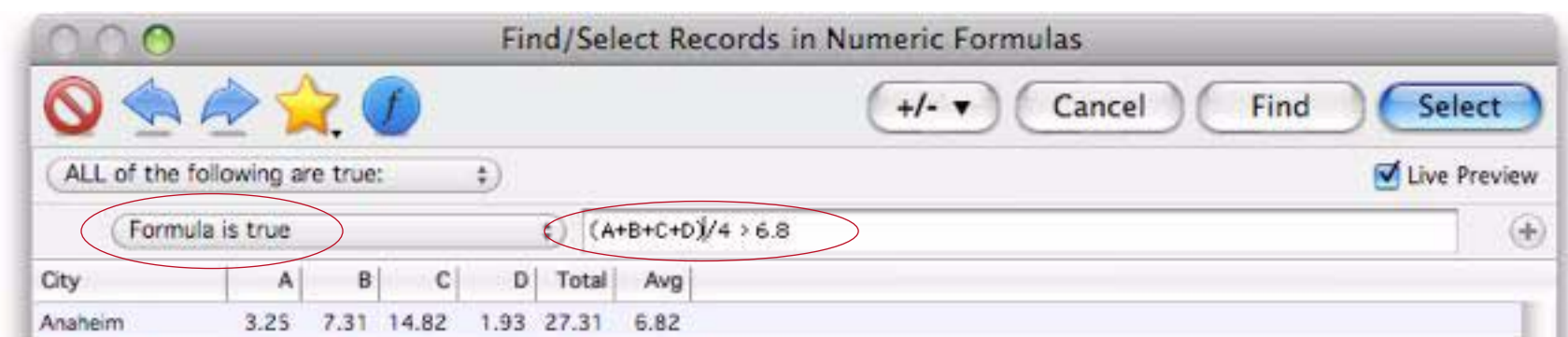
The **Total** and **Avg** fields will be updated any time the **A**, **B**, **C** or **D** fields are modified.

Using a Formula to Locate/Select Information

The **Find/Select** command (see “[The Find/Select Dialog](#)” on page 336) allows you to select data based on a formula. This allows you to make selections on data that is not directly stored in the database. For example, suppose you want to select all records with an average greater than **6.8**, but without actually storing the average in the database. Here’s the database.

City	A	B	C	D
Anaheim	3.25	7.31	14.82	1.93
Bakersfield	2.29	8.26	12.91	3.02
Camarillo	2.83	9.19	15.11	2.54
Diamond Bar	3.13	7.81	13.19	3.11
Fullerton	2.59	8.25	13.48	1.77
Laguna Beach	3.06	9.45	12.5	3.01
Newport Beach	3.18	7.22	12.32	2.38
Whittier	3.67	5.23	14.24	3.27

Choose the **Find/Select** command, then change the popup to **Formula is true**. Then enter the formula. This formula calculates the average and then compares the average to **6.8**.



When the **Select** button is pressed records with averages above the threshold are selected.

City	A	B	C	D
Anaheim	3.25	7.31	14.82	1.93
Camarillo	2.83	9.19	15.11	2.54
Diamond Bar	3.13	7.81	13.19	3.11
Laguna Beach	3.06	9.45	12.5	3.01

4 visible/8 total

Hey - I cheated (sort of)! This database already has the averages stored in the database. I can expand the window to double check that I really selected only records with averages over 6.8.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13	7.81	13.19	3.11	27.24	6.81
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00

4 visible/8 total

Do you forgive me? Anyway, the point is that the selection can be made even if the average is not stored in the database. Here's another example. This formula will select every record where the city name is longer than 10 characters (11, 12, 13, etc.)

Find/Select Records in Numeric Formulas

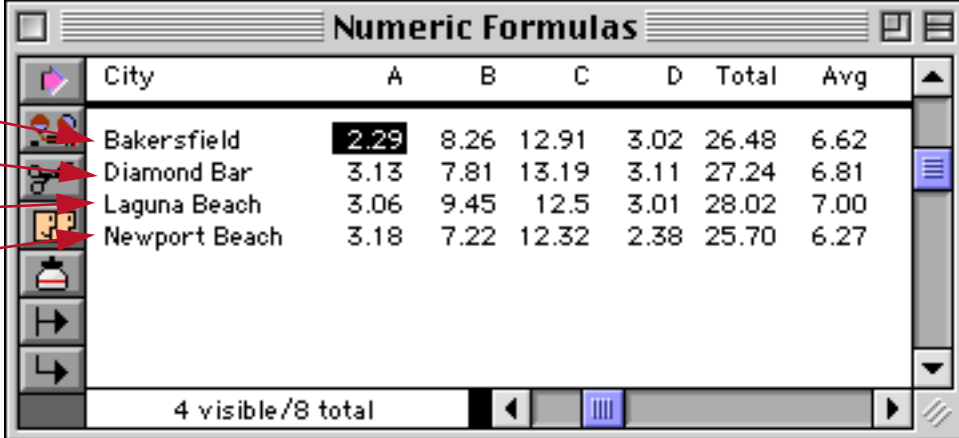
ALL of the following are true: Live Preview

Formula is true: `length(City) > 10`

City	A	B	C	D	Total	Avg
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Diamond Bar	3.13	7.81	13.19	3.11	27.24	6.81
Laguna Beach	3.06	9.45	12.50	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.10	6.27

4 matches/8 total

Press **Select** to see only the records with long city names.



City	A	B	C	D	Total	Avg
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Diamond Bar	3.13	7.81	13.19	3.11	27.24	6.81
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.70	6.27

With a bit of ingenuity you can almost come up with a formula to locate or select even the most obscure information.

Formulas in Procedures

Within a procedure formulas are used to calculate values and control program flow. Most procedures contain many formulas — a typical example is shown below. The formulas in this procedure (there are 25 visible in this window) have been highlighted with a light blue box. Don't worry, I don't expect you to understand this procedure right now — the point is to show how vital formulas are to the operation of almost any procedure, and to show the wide variety of formulas possible, from very simple like a single number or text item to a complicated multi-line formula.

```

local ACTION
ACTION=parameter(1)
case ACTION contains "start"
  disableabort
  fileglobal ProgressDescription,ProgressDetail,Boiling,ProgressAbortMode,
    ThermoWidth,Mercury,Temperature,oldTemperature,ProgressCount
  local dlgHeight,dlgWidth,dlgRectangle
  ProgressAbortMode="confirm"
  if info("trigger")="Abort"
    settrigger ""
  endif
  Boiling=parameter(2)
  ProgressDescription=parameter(3)
  ProgressDetail=""
  //call .OpenDialog,"Progress",78,236,"-pause"
  dlgHeight=292 dlgWidth=207
  if (dlgHeight*dlgWidth<(rwidth(info("windowrectangle"))*rheight(info("windowrectangle")))*0.5) and
    dlgWidth<rwidth(info("windowrectangle")) and dlgHeight<rheight(info("windowrectangle"))
    /* center dialog within current window */
    dlgRectangle=rectanglecenter(
      info("WindowRectangle"),
      rectanglesize(0,0,dlgHeight,dlgWidth))
  else
    /* center dialog within main monitor screen */
    dlgRectangle=rectanglecenter(
      info("ScreenRectangle"),
      rectanglesize(0,0,dlgHeight,dlgWidth))
  endif
  fitwindow
  setwindowrectangle dlgRectangle,"nopalette novertscroll nohorzscroll"
  openform "Progress"

  object "Thermometer"
  ThermoWidth=rwidth(objectinfo("rectangle"))
  SelectObjects ObjectInfo("name")="Mercury"
  Mercury=objectinfo("rectangle")
  oldTemperature=-1
  ProgressCount=0

```

To learn more about creating procedures see [“Procedures”](#) on page 203.

Using the Formula Wizard

Panorama includes a **Formula Wizard** that you can use as a “sandbox” for playing with formulas. The **Formula Wizard** lets you play around with formulas and see the results immediately. It’s great for trying out a new function or technique you are not familiar with or to work the bugs out of a formula before actually incorporating it into your database. To use this wizard start with any database and select Formula Wizard from the Calculation submenu in the Wizard menu. (Tip: Unless you’ve changed this setting with the **Hotkey** wizard, you can also open the Formula Wizard by pressing **Control-1** on Macintosh systems.)



To use the **Formula Wizard** type a formula into the top section.



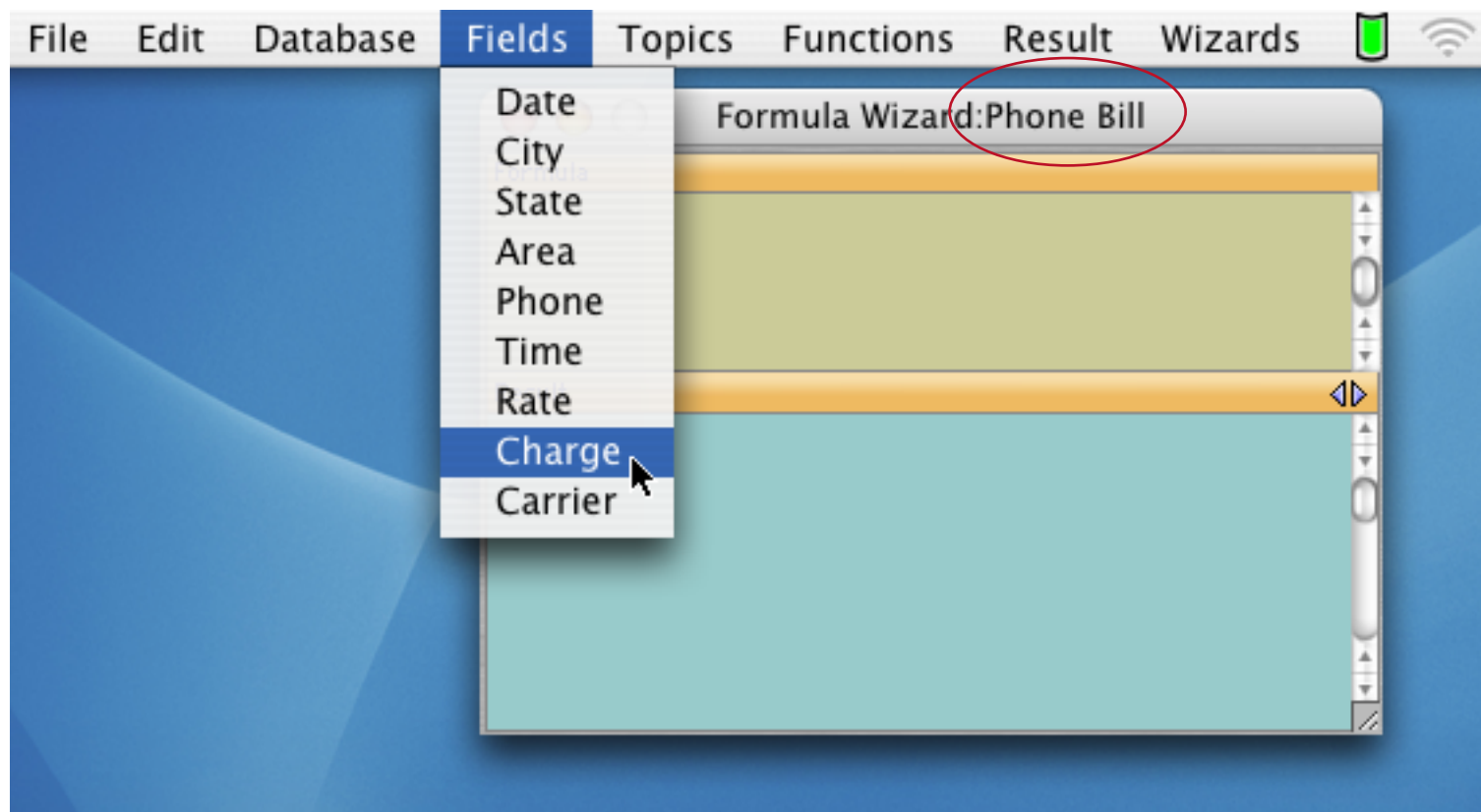
When you press the **Enter** key the result appears in the bottom section (see “[Arithmetic Formulas](#)” on page 60).



As you can see, the **Formula Wizard** can be used as a handy calculator.

Calculations with Database Fields

The **Formula Wizard** can include values in database fields as part of the calculation. The name of the database that is currently linked with the wizard is shown in the window's title bar, in this case **Phone Bill**. If you forget a field name you can see a list of all the fields in the **Fields** menu.



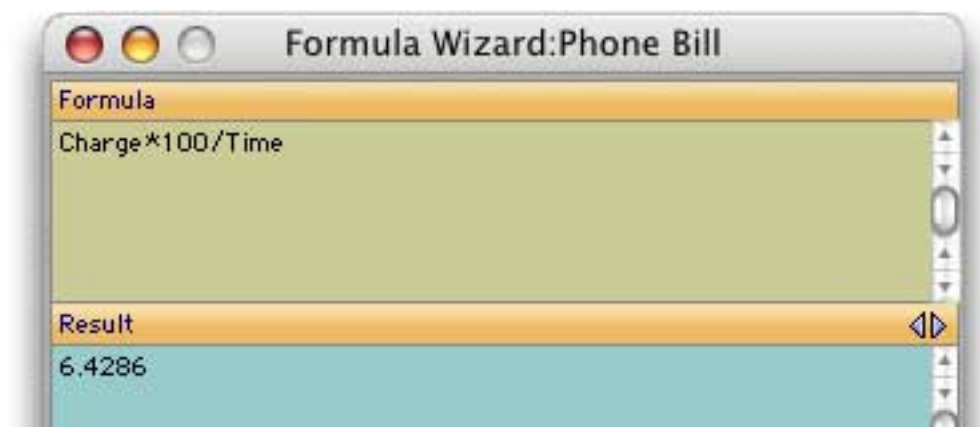
When you select a field from this menu the wizard will automatically type it in for you.



You can included as many fields as you want in the formula. However, they normally must all be from the active database, in this case **Phone Bill**.



Once again when you press the **Enter** key the wizard will calculate the result, in this case the cost of the phone call in cents per minute.

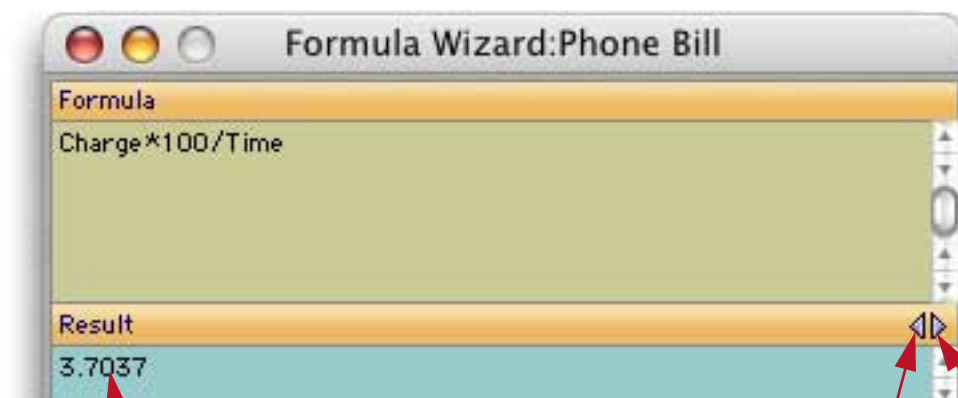


When the wizard performs the calculation it does so based on the currently active record in the active database. In this example the calculation was based on a charge of 2.70 for a 42 minute phone call.



Date	City	State	Area	Phone	Time	Rate	Charge	Carrier
01/17/84	Anaheim	CA	714	991-4328	42	DE	2.70	GTE
01/29/84	Anaheim	CA	714	533-6619	27	DN	1.00	GTE
01/29/84	Anaheim	CA	714	533-6619	63	DN	2.30	GTE
02/02/84	Anaheim	CA	714	533-6619	26	DE	1.69	GTE

You can use the two small triangles to try out the calculation for other records in the active database. As you press the triangles the active record will move up and down.



calculation automatically updates

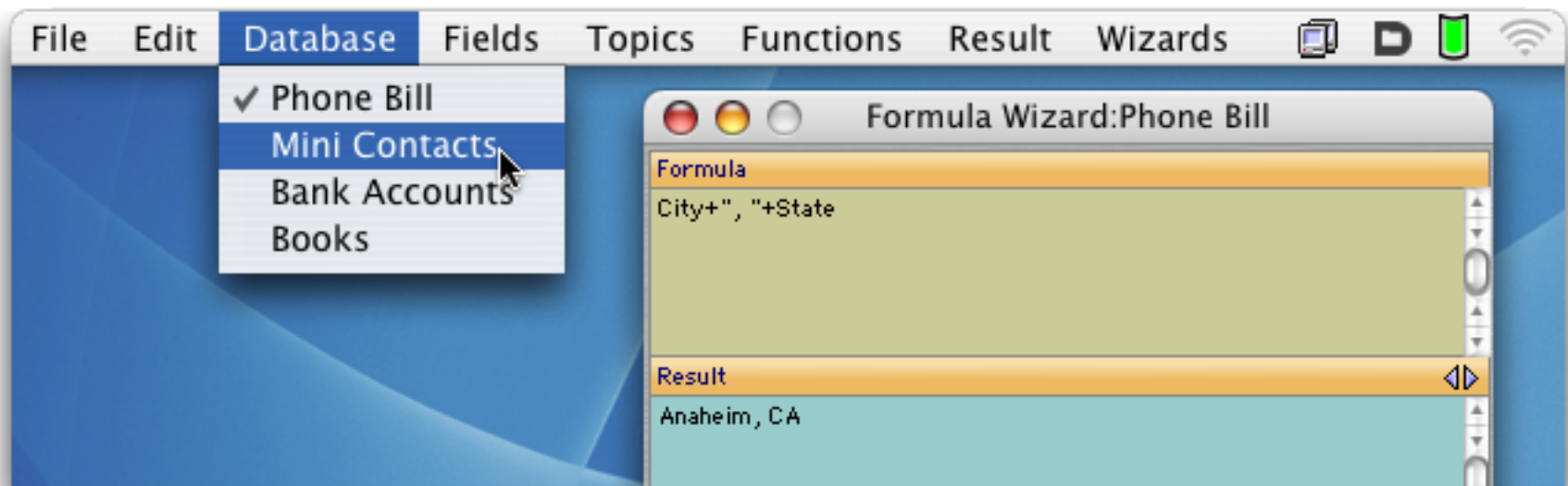
press to move down one record
press to move up one record

A formula can also work with text fields (see [“Text Formulas”](#) on page 67). This formula glues the city and state together with a comma in between (see [“Gluing Strings Together”](#) on page 67).



Changing the Active Database

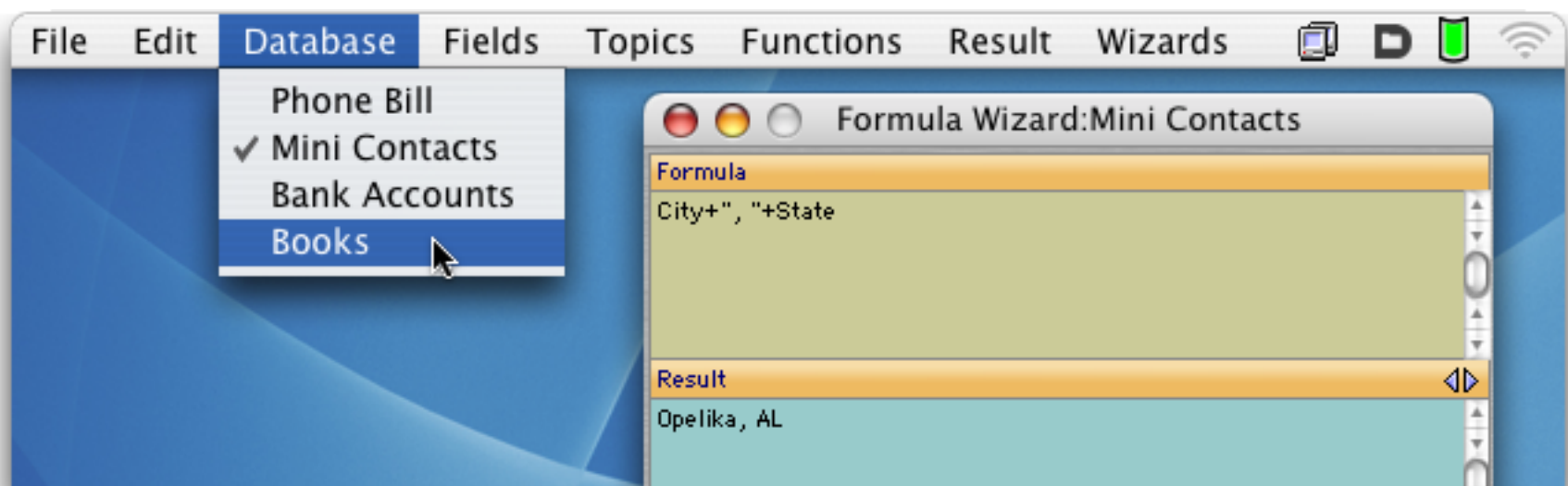
You can change the database that is used by the Formula Wizard at any time with the Database menu. This menu lists all of the currently open databases.



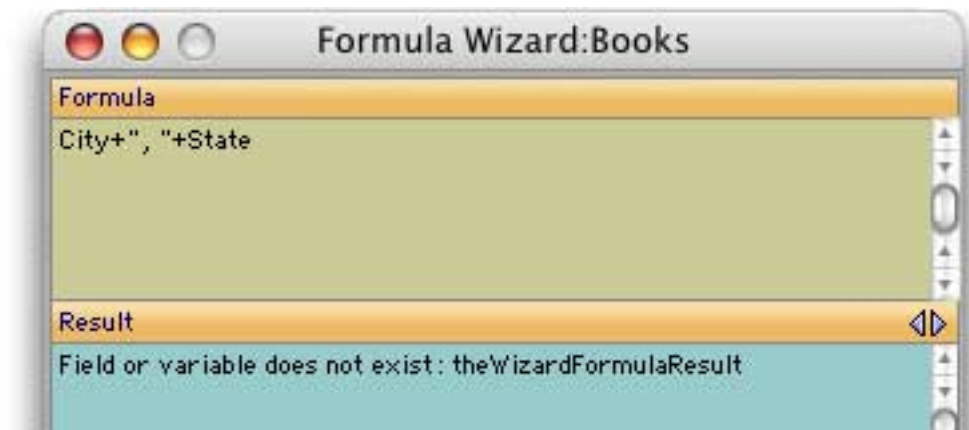
Now when we re-evaluate the same formula it comes up with a different result ([Opelika, AL](#)).



If you switch to a database that does not contain one or more of the fields used in the formula...



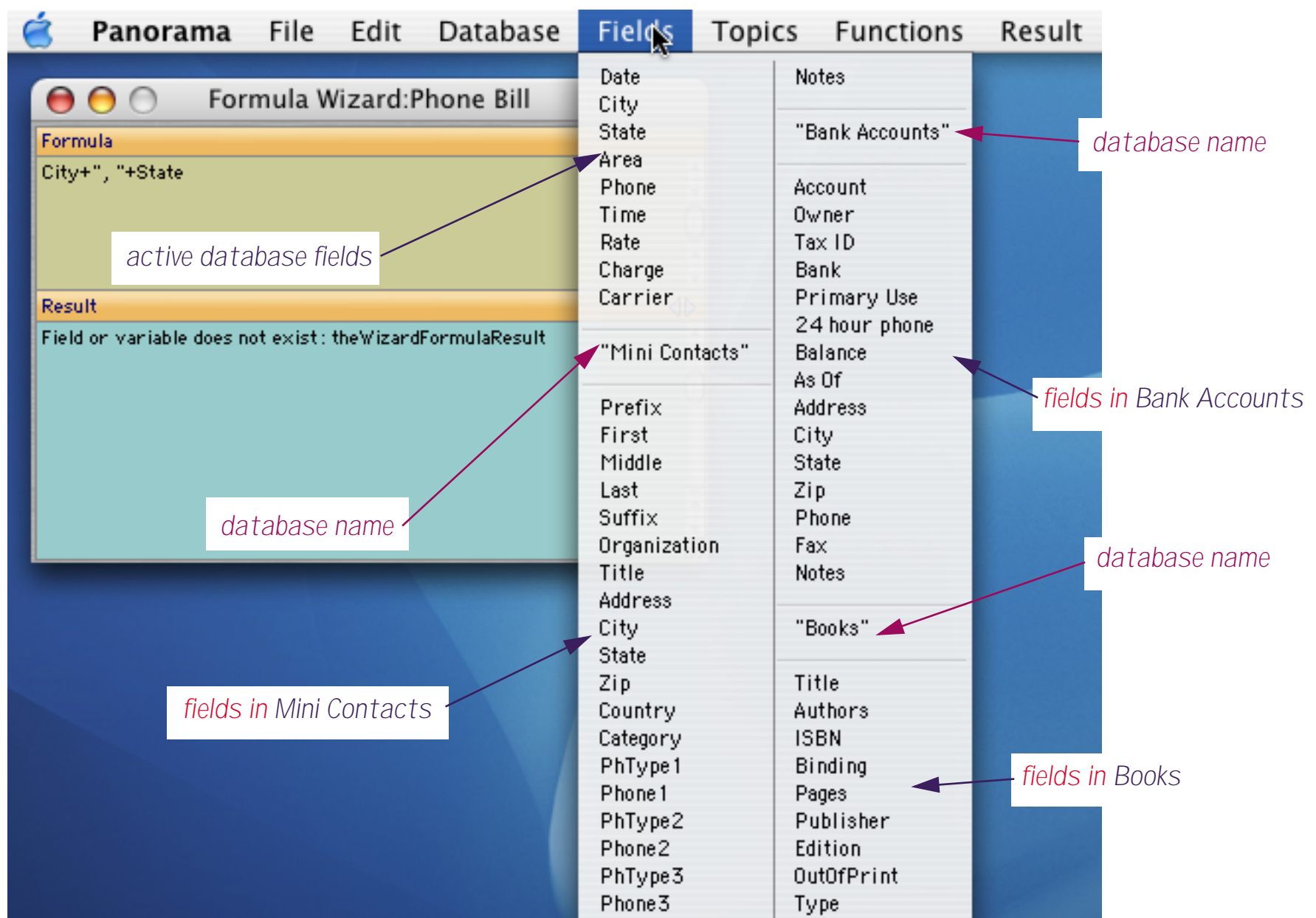
the result will be an error.



Another way to set the active database is to click on one of the database's windows and then choose **Formula Wizard** from the Calculations submenu of the Wizard menu. This brings the wizard back to the front and makes the selected database active.

Using Fields from Other Databases

Some functions (for example `lookup()` and `grabdata()`, see “[Linking With Another Database](#)” on page 131) use fields from other open databases, not just the active database. The **Fields** menu can help you type in these field names. When more than one database is open the **Fields** menu will list the fields in all of them. The fields for the active database (in this case **Phone Bill**) are listed first, followed by the name of each database (in quotes) and the fields for each database.



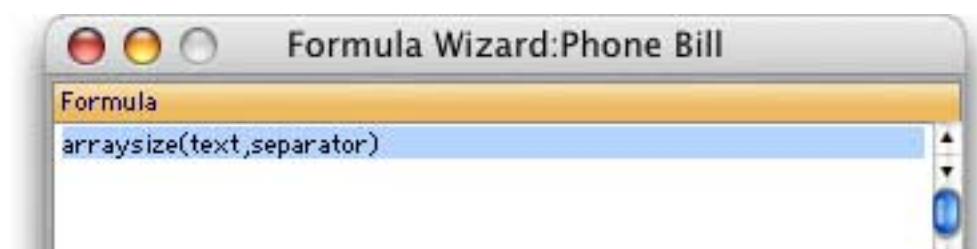
To type in a database name or field simply select it from the menu.

Topic and Functions Help Menus

Panorama contains hundreds of function and operators available for your use. Even here at ProVUE Development we can't remember all of them, so we have several methods for finding a particular function. The **Topics** menu organizes functions into submenus by topic.



When you release the mouse the function or operator will be typed into the formula.



If the function has one or more parameters (see “[Multi-Parameter Functions](#)” on page 44) you can select the first parameter by pressing **Command-1** (Macintosh) or **Control-1** (Windows PC). This selects the first parameter of the function.

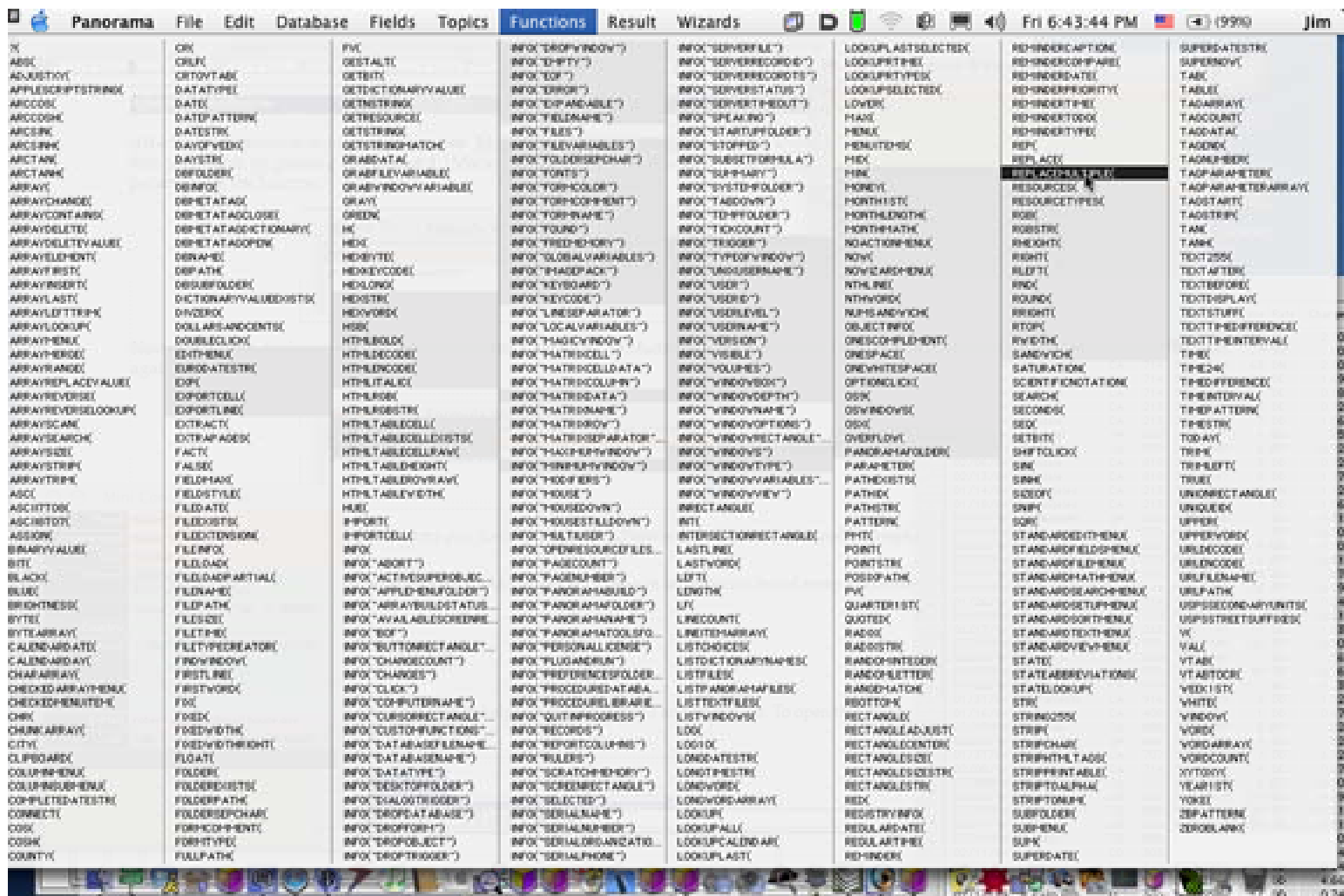


Now you can type in the actual parameter, then press **Command-1** (Macintosh) or **Control-1** (Windows PC) again to advance to the next parameter.



If this function had additional parameters you could continue to advance until all of the parameters were complete.

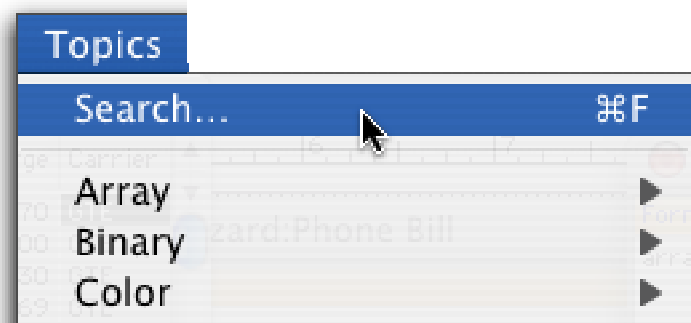
The **Topics** menu lists most, but not all of Panorama’s functions. To see an alphabetical list of every available function use the **Functions** menu. As you can see, this is a giant menu that will fill your entire screen.



Just like the **Topics** menu, you can select any function in this menu and it will be typed into the Formula Wizard for you.

Function Search

The Search dialog provides an alternate way to help you enter functions and operators. To open this dialog choose **Search** from the Topics menu.



The dialog displays an alphabetized list of functions and operators.

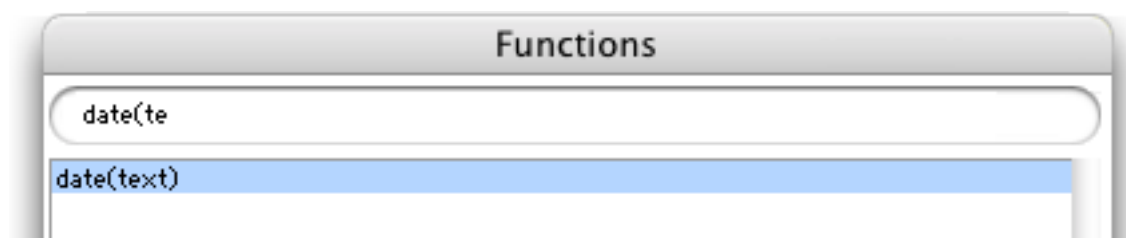


To type a function or operator into the formula you can click on it and press **OK** or the **Enter** key, or you can simply double click on the function or operator you want.

The list initially contains several hundred functions and operators. To cut down this list you can type in a search criteria. As you enter each key the number of items in the list will be reduced to show only functions or operators that contain the text you have typed in. For example the list below shows only functions that contain the word [date](#).



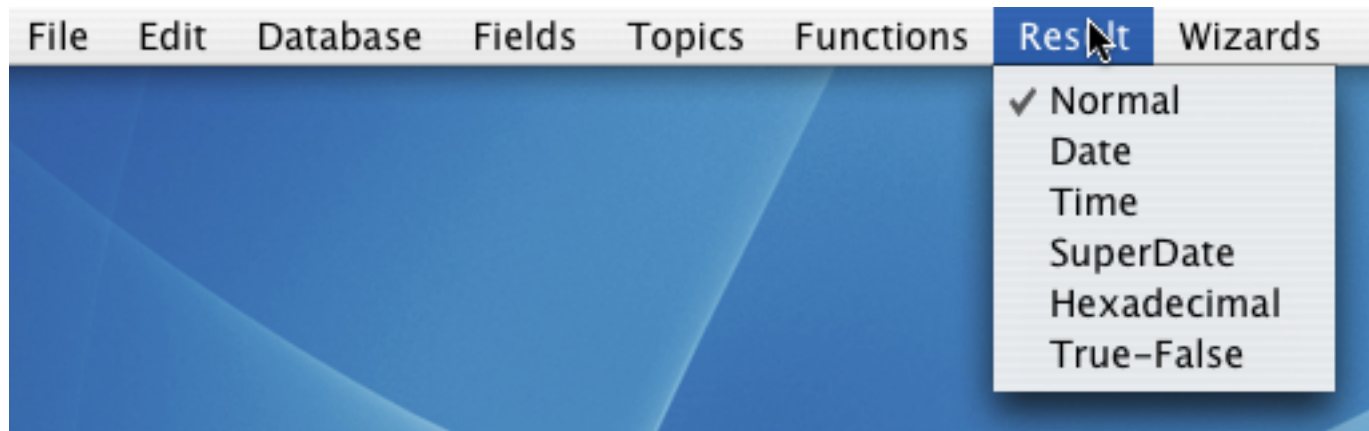
At any time you can click on one of these functions and press **OK** or the **Enter** key, or you can simply double click on the function you want. If the list has been reduced to a single item you can simply press the **Enter** key.



After the function or operator has been typed in you can advance through the parameters as described in the previous section.

Special Formula Result Formats

The Formula Wizard normally displays the result of the formula as a number or as text. Using the **Formula** menu you can select other custom formats for displaying the result.



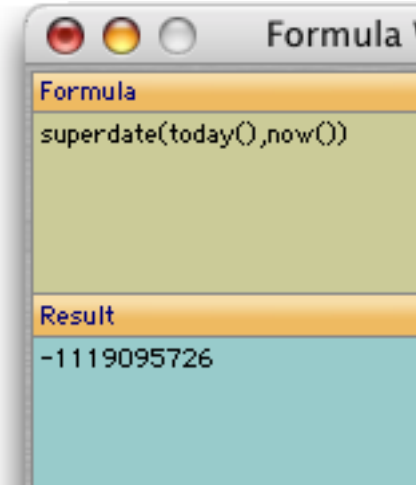
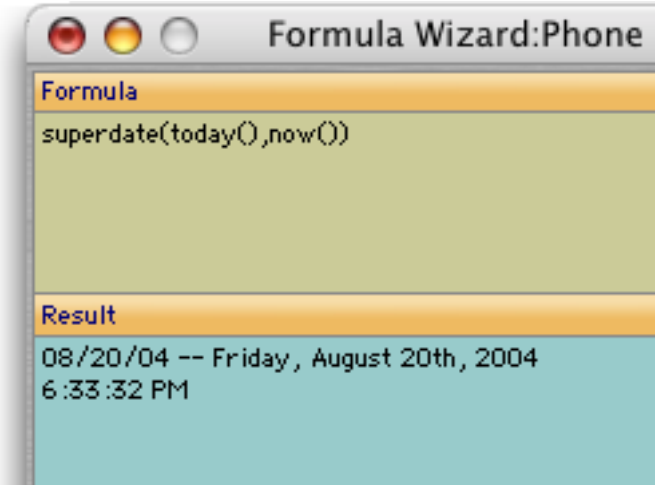
Use the **Date** option when you want to display the numeric result of a formula as a date (see “[HTML Generating Functions](#)” on page 105). The table below illustrates how a date is displayed with both the **Normal** and **Date** options.

Normal	Date
<p>Formula today() Result 2453238</p>	<p>Formula Wizard:Phon Formula today() Result 08/20/04 -- Friday, August 20th, 2004</p>

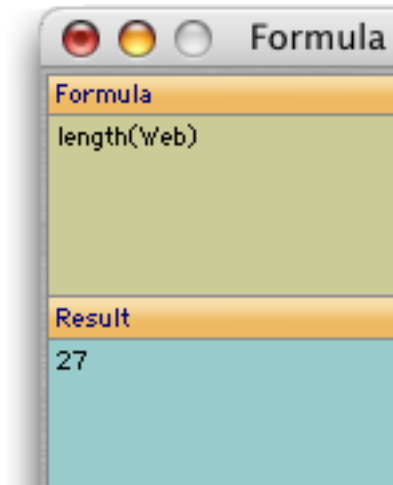
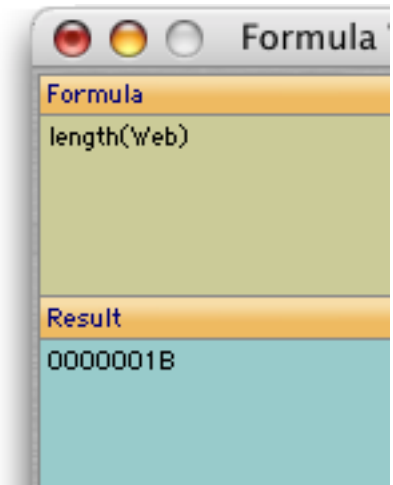
Use the **Time** option when you want to display the numeric result of a formula as a time (see “[Time Arithmetic](#)” on page 113). The table below illustrates how a time is displayed with both the **Normal** and **Time** options.

Normal	Time
<p>Formula now() Result 66638</p>	<p>Formula Wizard: Formula now() Result 6:31 :24 PM</p>

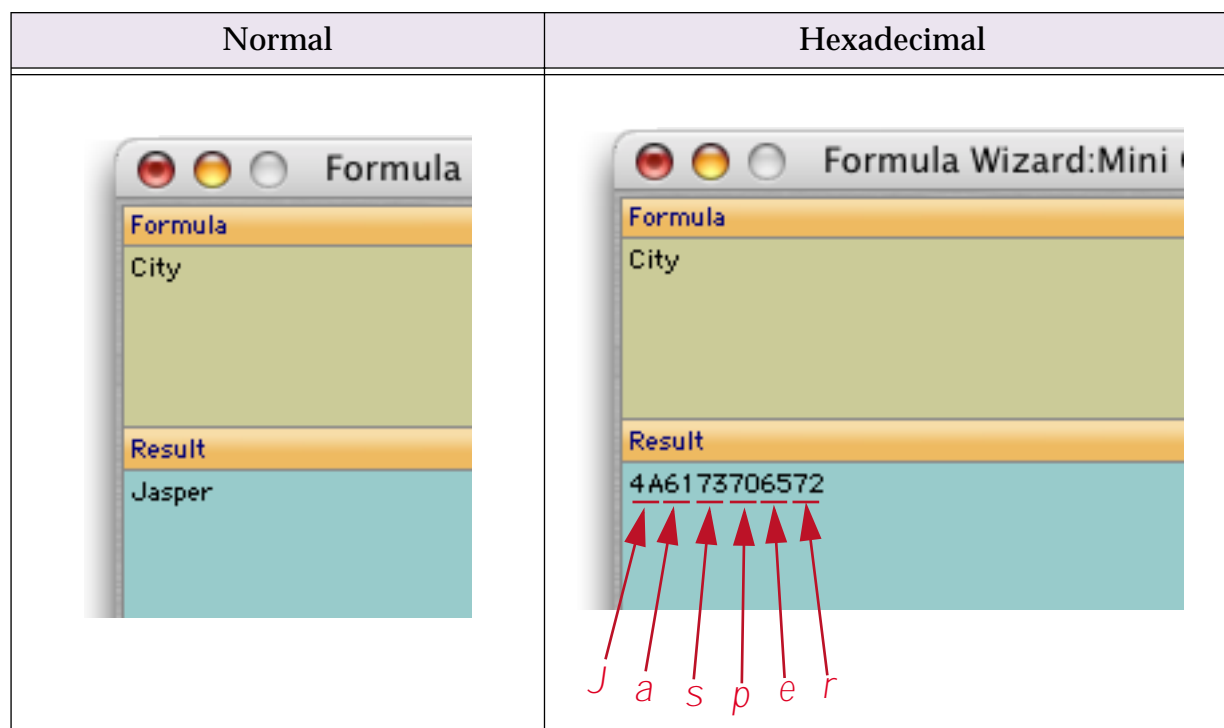
Use the **SuperDate** option when you want to display the numeric result of a formula as a SuperDate (see “[SuperDates \(combined date and time\)](#)” on page 118). The table below illustrates how a result is displayed with both the **Normal** and **SuperDate** options.

Normal	SuperDate
 <p>Formula superdate(today(),now())</p> <p>Result -1119095726</p>	 <p>Formula Wizard:Phone Formula superdate(today(),now())</p> <p>Result 08/20/04 -- Friday, August 20th, 2004 6:33:32 PM</p>

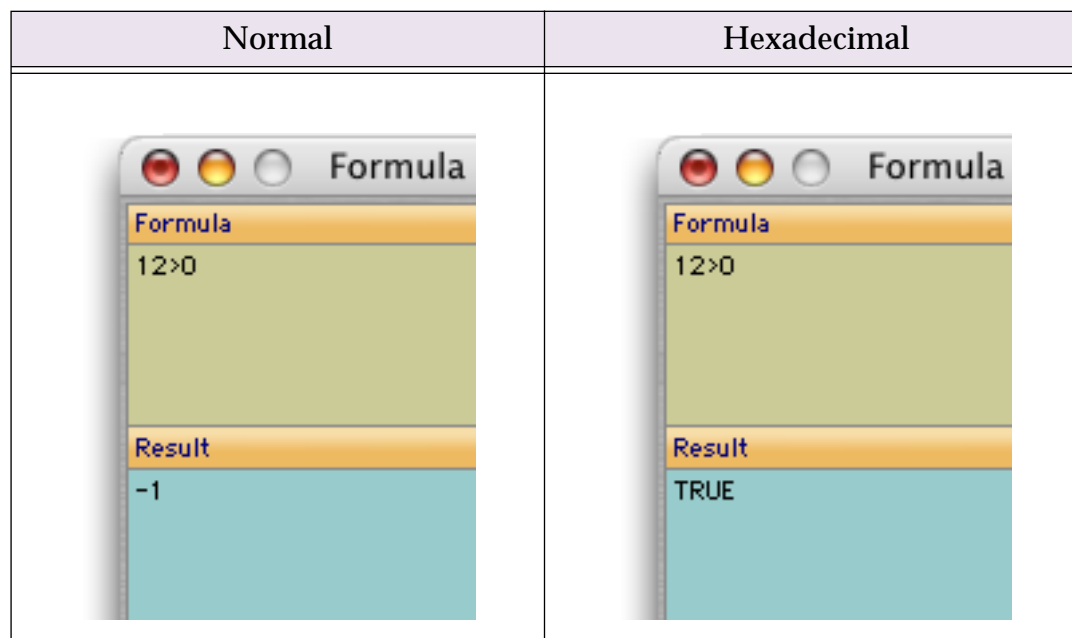
Use the **Hexadecimal** option when you want to display the result of a formula as a hexadecimal number (see “[Raw Binary Data](#)” on page 156). The table below illustrates how a number is displayed with both the **Normal** and **Hexadecimal** options.

Normal	Hexadecimal
 <p>Formula length(Web)</p> <p>Result 27</p>	 <p>Formula length(Web)</p> <p>Result 0000001B</p>

The **Hexadecimal** option may also be used to display text results. The ASCII value of each character is displayed in hexadecimal.

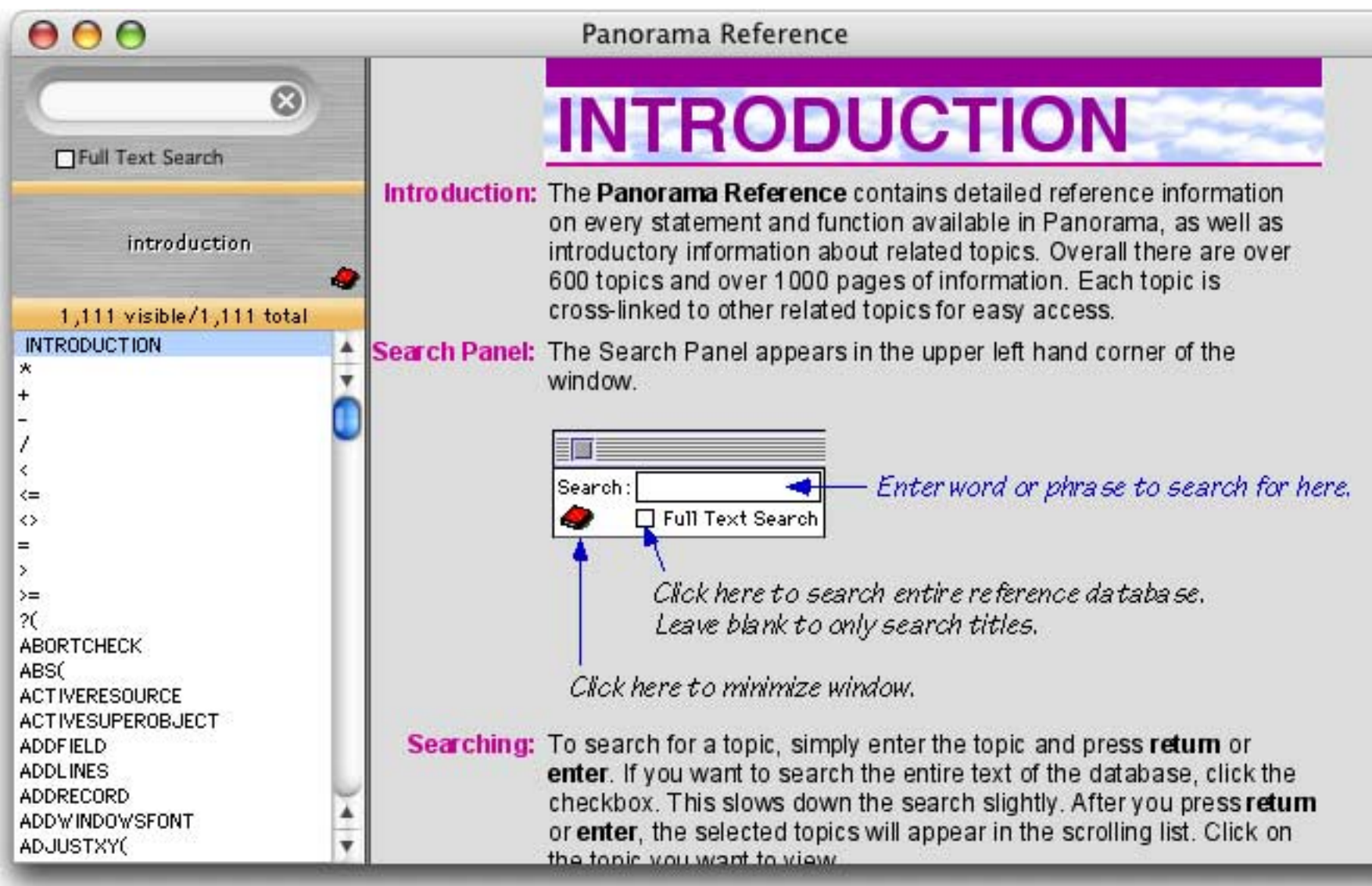


Use the **True-False** option when you want to display the result of a formula as true or false (see “[True/False Formulas](#)” on page 124). The table below illustrates how a number is displayed with both the **Normal** and **True-False** options.

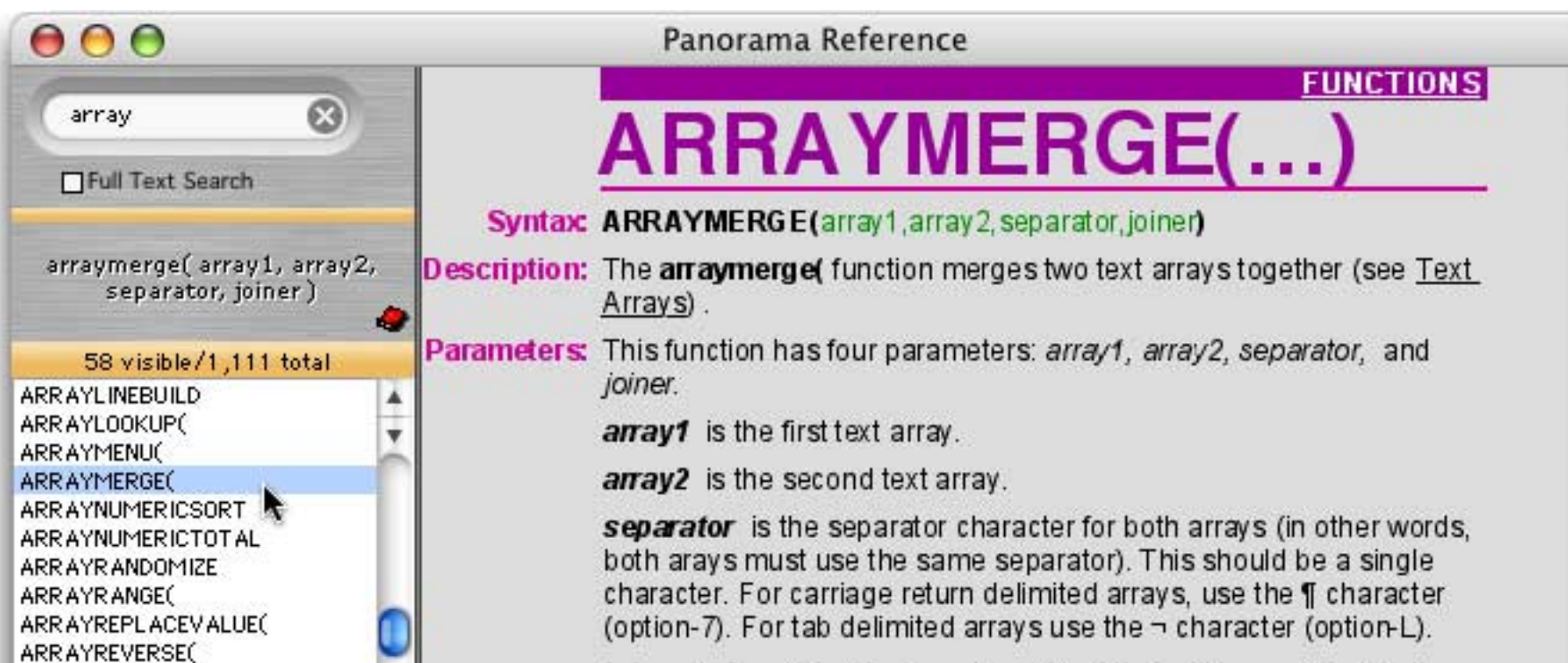


The Programming Reference Wizard

The fastest way to find complete information for any function or operator is to use the **Programming Reference** wizard. To open this wizard, select it from the Documentation submenu of the Wizard menu, or simply press **Control-R** if you are using a Macintosh computer.



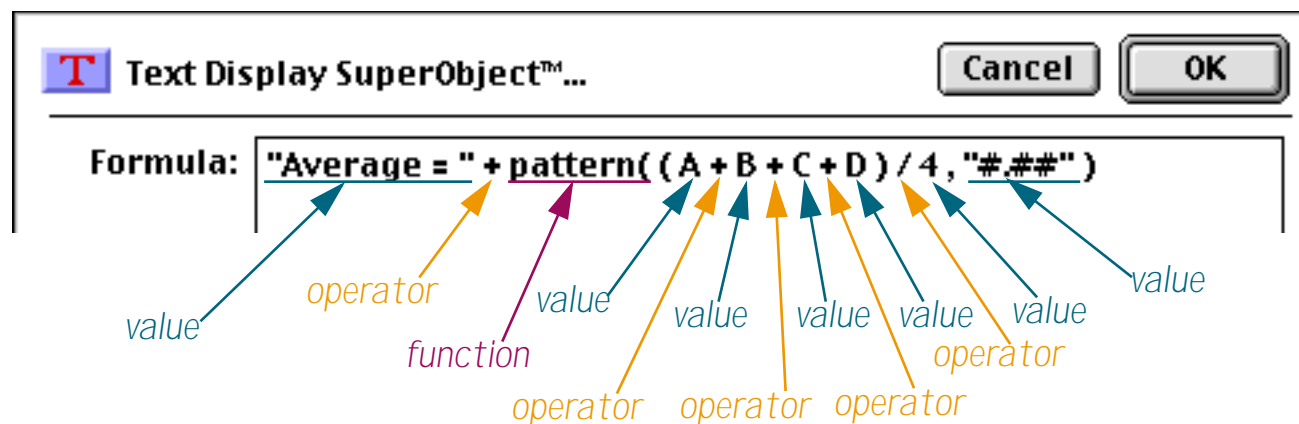
To find complete information for any function or operator simply type in the name of the function or operator in the search box in the upper left, and/or select the function or operator in the scrolling list on the left.



For additional information on this wizard see [“Programming Reference Wizard”](#) on page 237.

Formula Components

Just as a sentence is constructed from basic words, a formula is created by combining simple elements — **values** (also called **operands**), **operators** and **functions**. Values (operands) are roughly equivalent to nouns, while operators and functions act as verbs. This illustration shows the components that go into a typical formula.



Formula Grammar

Panorama formulas have grammar rules just as languages like English and Spanish do. These rules tell how values, operands and functions can be combined to make a valid formula.

The simplest formula is a single data value. Here are four examples of such simple formulas.

`A`

`47`

`"Oregami"`

`ShippingMethod`

Two values can be combined with an operator in between. The first example below adds two numbers together. The second example multiplies two numbers together. The third example appends two text values together (to produce a value like [Mr. Jones](#)).

`2 + 2`

`Total * TaxRate`

`"Mr. " + LastName`

The values must be the appropriate type for the operator. For example, you can multiply two numbers together like this

`2 * 2`

but you cannot multiply two text values together like this (see "[Grammar Errors](#)" on page 47).

`"Mr. " * LastName`

You can combine three or more data values with an operator between each pair of values.

`7 + 3 * 4 / 2`

`FirstName + " " + MiddleInitial + " " + LastName`

Calculation Order and Parentheses

When a formula contains more than one operator, the calculations are performed from left to right unless one of the operators has a higher precedence (priority). This is the natural arithmetic order—multiply and division first, then addition and subtraction. This table lists the order of precedence for all operators.

1. Unary minus (example: -12)
2. Raise to power (example: 10^5)
3. Multiply and Divide
4. Integer Divide
5. MOD (remainder)
6. Add and Subtract
7. Comparisons (=, <>, <, >, ...)
8. NOT
9. AND
10. OR and XOR

For example, consider the formula below.

$$7 + 3 * 4 / 2$$

Panorama first multiplies $3 * 4$ to get **12**, then divides this by **2** to get **6**. Finally it adds **7** (addition is last because of its low precedence) to get the final result, **13**.

You can override the natural calculation order with parentheses. For example, the parentheses in the formula below force the addition to be calculated first, then the multiplication and division.

$$(7 + 3) * 4 / 2$$

Now the final result is **20** instead of **13**. When in doubt you can always add parentheses to force Panorama to calculate the formula in any order you want.

Functions

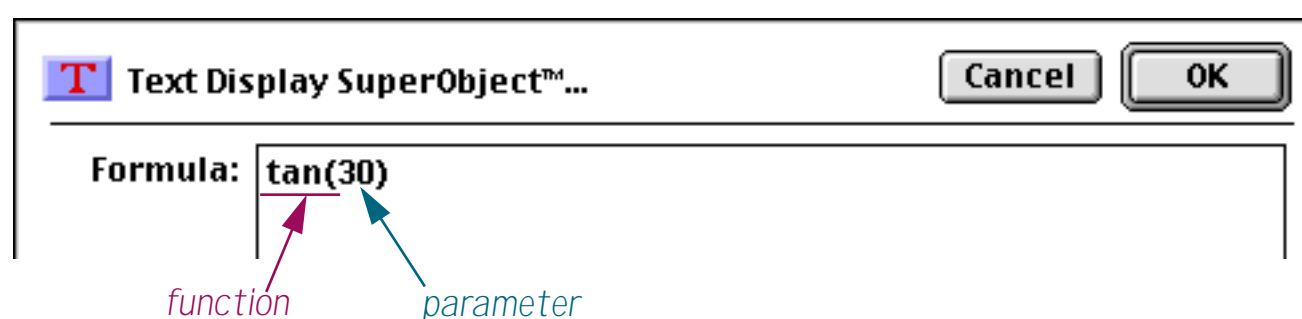
A function is a formula component that calculates a value. It may calculate the value out of thin air (for example, calculating the value of the current date or time) or it can calculate the value from other values (for example trigonometry functions calculate values from angles). Panorama has several hundred functions available. Each function has a name, and is always followed by parentheses. For example, the **tan()** function calculates the tangent (a trigonometry function) of an angle.

$$\tan(30)$$

A function can be used in a formula anywhere a regular value can be used. Just as with ordinary values, you can use operators to combine functions with other values (and functions).

$$3 + \tan(30)$$

The value operated on by the function is called a **parameter**.



A function takes the parameter value (in this case **30**) and transforms it into another value (in this case **-6.4053**, the tangent of **30**). The parameter can be a formula itself, like this.

```
tan( A + B )
```

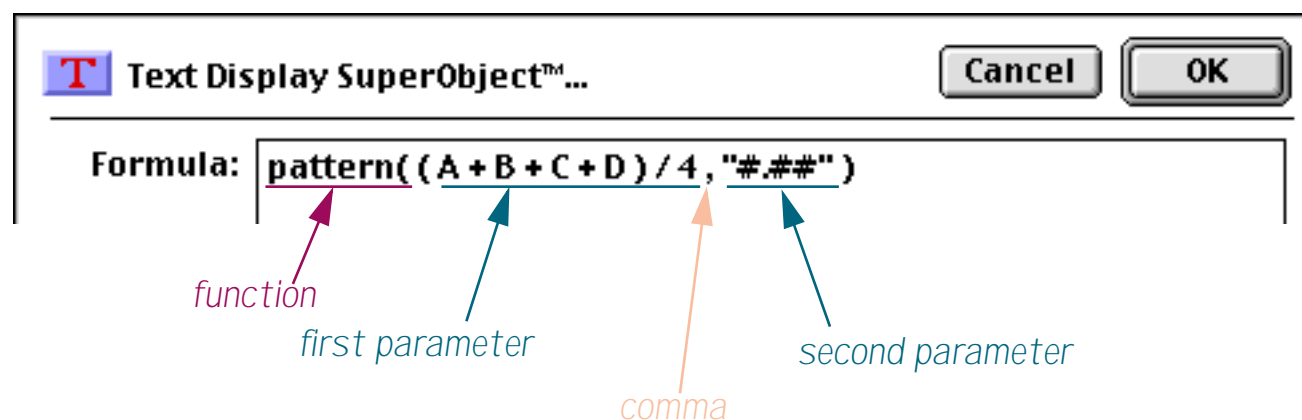
In this case Panorama first calculates the value **A+B**, then computes the tangent of that sum. A parameter may be as complex a formula as you need, with additional parentheses and even other functions nested inside the first function.

```
tan( sqr( A + B ) + 1 )
```

The parameter to the **sqr()** function is **A+B**, while the parameter to the **tan()** function is **sqr(A+B)+1**. (The **sqr()** function, by the way, calculates square roots.) Panorama will always calculate the formula from the inside out until the entire formula has been computed.

Multi-Parameter Functions

Many functions use more than one parameter. When more than one parameter is required each parameter is separated from the next by a comma. All of the parameters are surrounded by parentheses, just as with single parameter functions. For example, the **pattern()** function (shown below) requires two parameters. The first parameter must be a numeric value (in this case a calculated average) and the second parameter must be a text value containing a pattern for formatting the number (see “[Numeric Output Patterns](#)” on page 250 of the *Panorama Handbook*).



Some functions require as many as six parameters. You must always supply every parameter — you cannot leave one out (see “[Grammar Errors](#)” on page 47).

Zero Parameter Functions

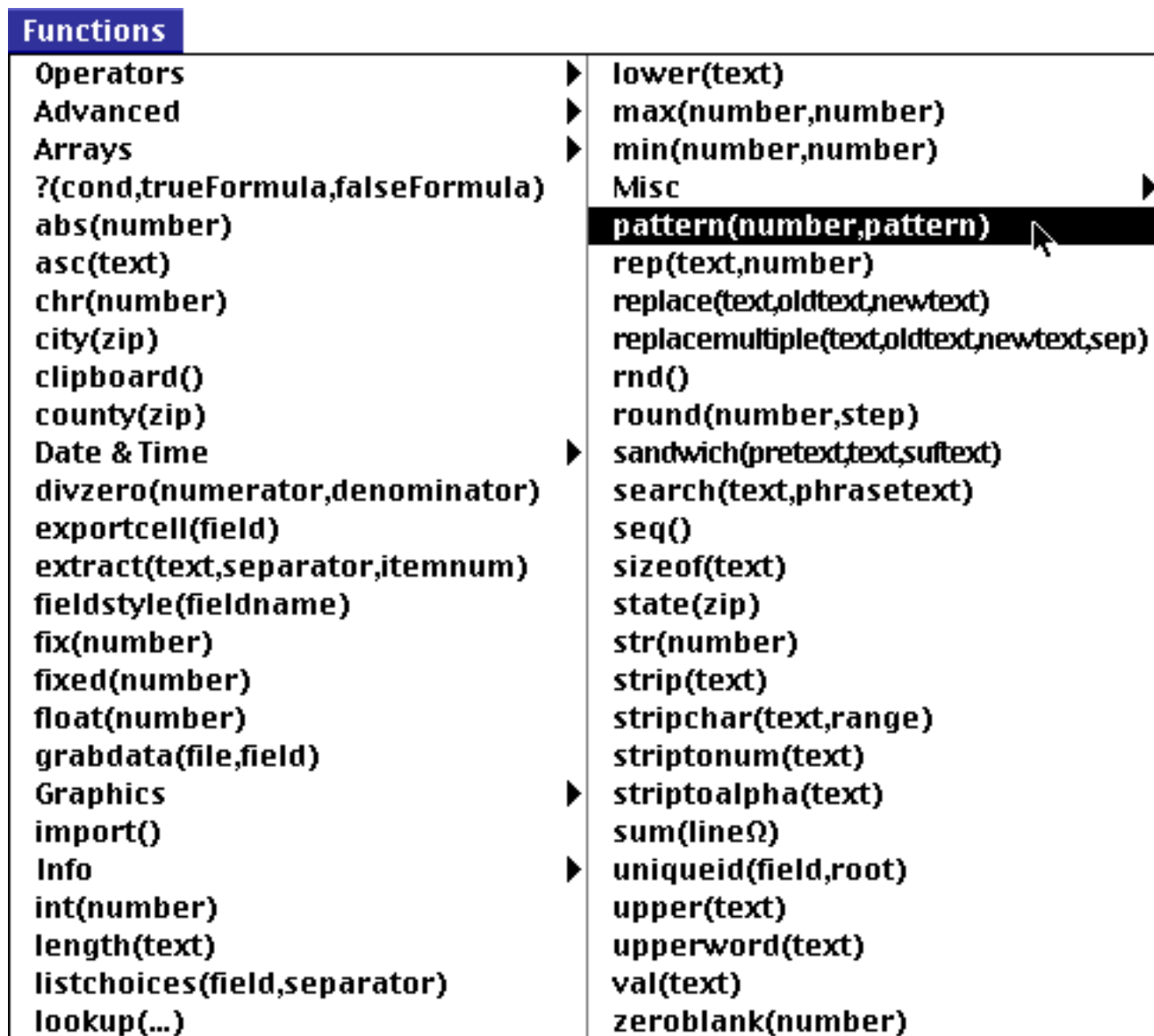
A small handful of functions don't require any parameters at all. These functions generate a value all by themselves, either by consulting the computer hardware (current date, current time), querying internal Panorama data (line number, imported data) or by generating a completely random number each time the formula is computed.

```
today()      -- current date
now()        -- current time
seq()        -- line number
import()     -- line of text from import file
rnd()        -- random number
```

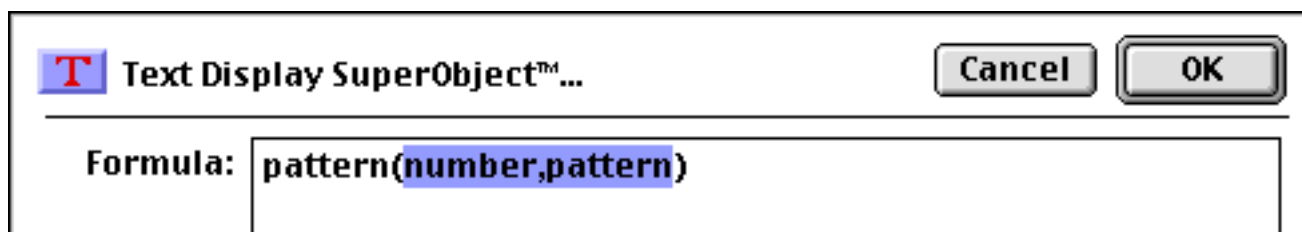
As you can see, these functions simply have both parentheses next to each other, with no parameter in between. You cannot omit the parentheses — you are required to include them as shown in the examples above.

Functions Menu

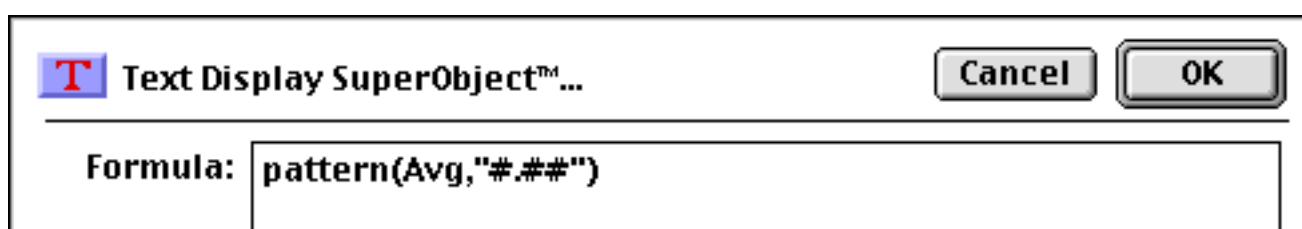
To help you type in a formula without errors Panorama has a special **Functions** menu that will type in a function for you. This menu is available whenever you enter a formula into a dialog, the design sheet or a procedure window. The function has a number of submenus that allow you to select from many of the hundreds of functions available. (Note: The Functions menu does not have ALL of Panorama's functions. For a complete and always up-to-date list see the Formula Wizard (see "[Topic and Functions Help Menus](#)" on page 34) and the Programming Reference Wizard (see "[The Programming Reference Wizard](#)" on page 41).)



When you select the function you want Panorama will automatically type it in for you. If the function requires parameters Panorama will type in a template for each parameter.



Simply type in the actual parameters to complete the function.



Whitespace

Most of the examples you've seen so far have extra spaces between the components, like these.

```
7 + 3 * 4 / 2
```

```
FirstName + " " + MiddleInitial + " " + LastName
```

```
tan( sqr( A + B ) + 1 )
```

Panorama ignores spaces between components. You can leave out the spaces, like this.

```
7+3*4/2
```

```
FirstName+" "+MiddleInitial+" "+LastName
```

```
tan(sqr(A+B)+1)
```

Or you can add extra spaces between components, or even carriage returns, like this. (Note: Some dialogs do not allow you to enter carriage returns, because pressing the **Return** key closes the dialog.)

```
7 + 3 * 4 / 2
```

```
FirstName + " " +
MiddleInitial + " " +
LastName
```

```
tan(   sqr( A + B ) + 1   )
```

Spaces are only ignored **between** components, not within components. A common mistake is to place a space in between the function name and the left parenthesis. This is not allowed. The formula below will not work (see "[Grammar Errors](#)" on page 47) because of the spaces after **tan** and **sqr**.

```
tan ( sqr ( A + B ) + 1 )
```

Another common problem is spaces or other punctuation in field names. If your database has fields named **First Name**, **Middle Initial** and **Last Name** you might be tempted to try a formula like this.

```
First Name + " " + Middle Initial + " " + Last Name
```

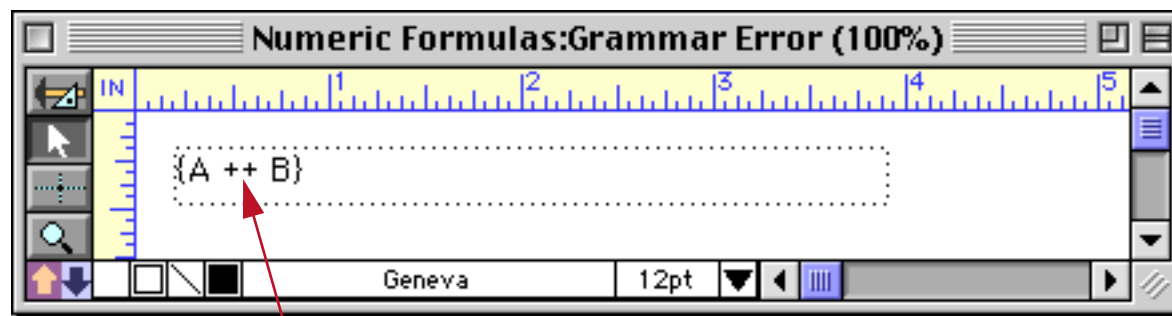
Sorry, but it won't work (see "[Grammar Errors](#)" on page 47). Because of the spaces inside the field names, Panorama will think that **First** and **Name**, **Middle** and **Initial** and **Last** and **Name** are separate components. The solution is to place chevron (« and ») characters around the field names. In many cases you can use the **Field** menu to type in the field name with chevrons for you. Otherwise, on the Macintosh press **Option-\<** to create the « chevron character and **Shift-Option-\<** to create the » chevron character. On Windows systems press **Alt-0171** to create the « chevron character and **Alt-0187** to create the » chevron character. Here's the revised formula, which will work perfectly

```
«First Name» + " " + «Middle Initial» + " " + «Last Name»
```

You'll also need to put chevrons around a field or variable name that contains punctuation, for example **«P/E Ratio»**. Without the chevrons Panorama will think that this is four separate components — **P**, **/**, **E** and **Ratio**.

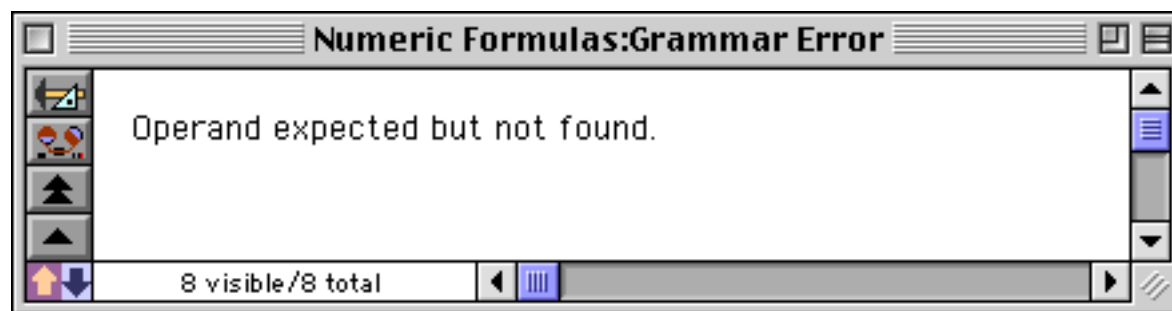
Grammar Errors

Unlike a human listener, Panorama is not able to tolerate incorrect or sloppy grammar. If you ask Panorama to calculate a formula that has incorrect grammar it will refuse to comply until you correct the mistake. For example, consider the formula shown below in an auto-wrap text object.



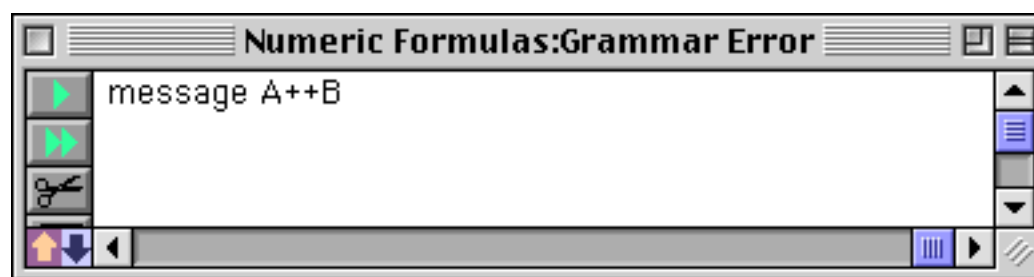
Grammar error! Can't have two operators in a row!

When you switch to Data Access Mode Panorama tells you about the grammar error.

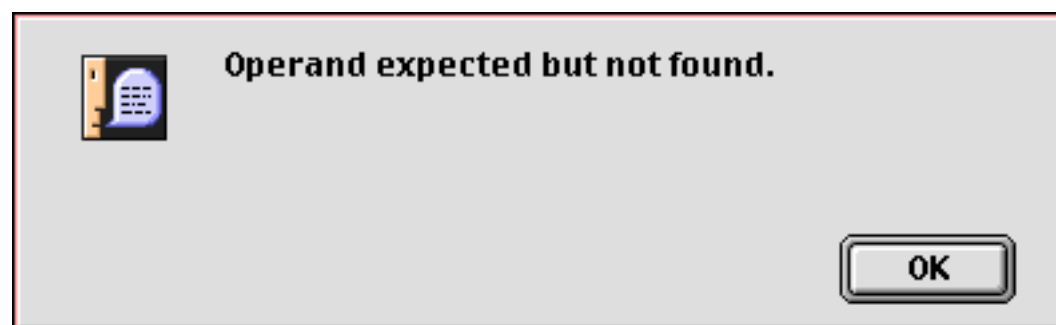


The error message tells you that Panorama expected an operand (value) after the + operator. The solution is either to remove the extra + operator or add another value in between the two + symbols.

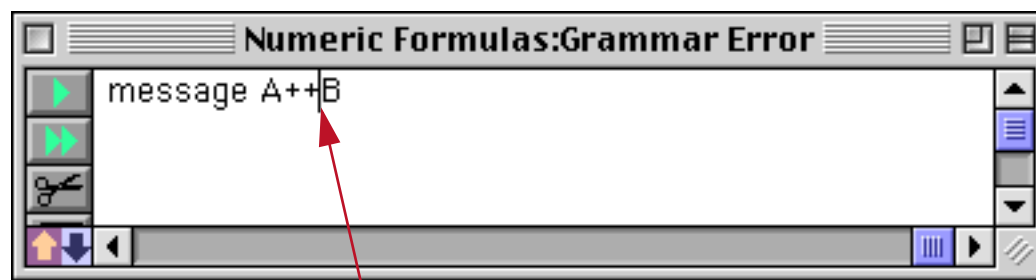
When you are editing a formula within a procedure, Panorama will attempt to point out the location of the grammatical error. For example, here is the same formula with the same error used in a procedure.



If you click to another window or use the **Check Procedure** command (in the Edit menu) Panorama will display an alert letting you know about the problem.

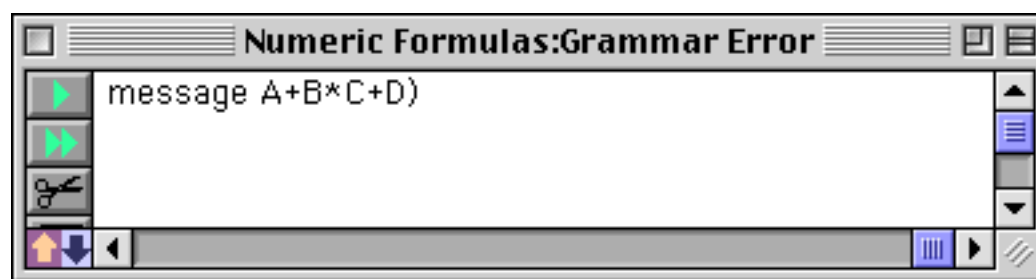


When you close the alert window Panorama will move the insertion point to the location where Panorama detected the error.

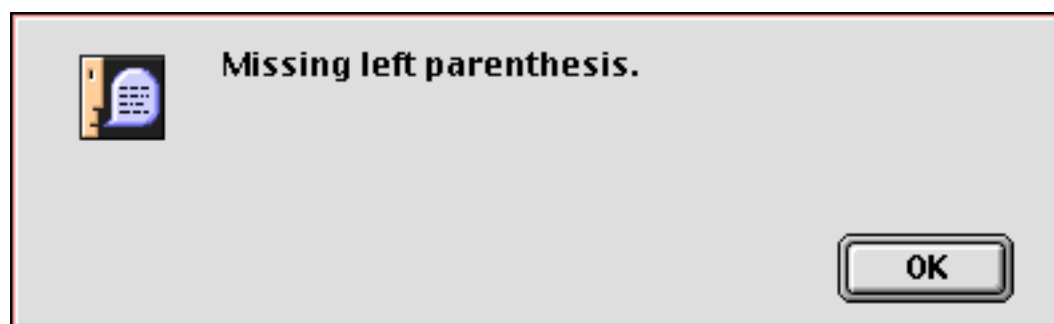


location where Panorama detected the error

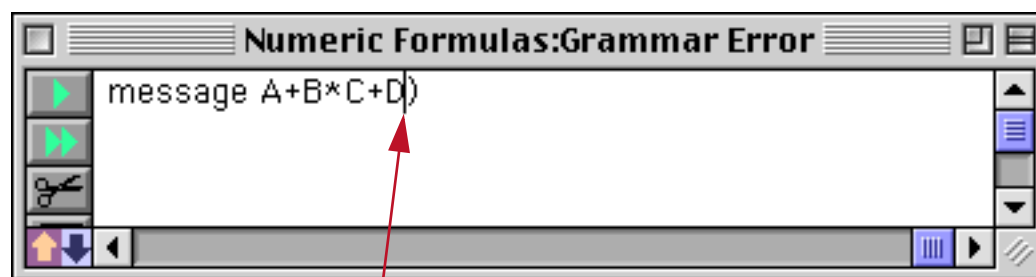
This location is usually fairly close to where the actual error is. However, in some cases Panorama is unable to determine exactly where the problem is. Consider the formula shown below, which has a missing left parenthesis.



When you click to another window or use the **Check Procedure** command (in the Edit menu) Panorama will display an alert letting you know about the problem.



When you close the alert window Panorama will move the insertion point to the location where Panorama detected the error.



location where Panorama detected the error

But wait — is this really where the error is? No, the error actually is somewhere earlier in the formula. In this case the missing (probably goes in front of the **B** or the **C**. Panorama has done the best job it could to locate the error for you. One thing you do know for sure, though, is that the error is always before the insertion point and not after.

Comments

Panorama allows “comments” to be placed inside a formula. A comment is a note within the formula that is ignored when the formula is evaluated. A comment must start with `/*` and end with `*/`. Anything between these will be ignored. (C and JavaScript programmers will recognize this style.) For example, this formula:

```
<CRatio> /* CRatio must be updated every 24 hours */ * Amount
```

will produce the same value as this one:

```
<CRatio> * Amount
```

In addition to notes to yourself, comments are also useful for temporarily disabling a section of a long formula (for example if you are trying to debug the formula).

Values

Values are the raw material that formulas work with—numbers and text. A value may be embedded in the formula itself, may be stored in a database field or may be stored in a variable (see “[Variables](#)” on page 53 and “[Variables](#)” on page 247).

Constants

When a value is embedded in the database itself it is called a **constant**. A numeric constant may be in fixed point format, like the numbers in this example (the numeric constants are highlighted in purple).

```
x + 2
```

```
today() - 90
```

```
Total * 0.0625
```

A numeric constant may also be in floating point format, which consists of the mantissa followed by the letter **e** followed by the exponent. The example below is equivalent to the mathematical formula $x \cdot 6.02^{23}$.

```
x * 6.02e23
```

A formula may also contain **text constants**. A text constant is a series of characters surrounded by quotes. When writing a text constant you may choose from five different types of quotes, as shown in this table.

Type	Open	Close	Example
Double Quote	"	"	"January"
Single Quote	'	'	'Tuesday'
Curly Braces	{	}	{San Francisco}
Smart Double Quote	“	”	“Gothic”
Smart Single Quote	‘	’	‘Bohemian’
Pipes	, , etc.	, , etc.	abc

The primary reason for different types is to allow quotes themselves to be used in a text constant. Suppose that you needed to use the text **The shim was 6" high** in a formula. Using double quotes around the constant will cause a grammar error, because Panorama won’t know what to do with the text after **6"** (shown in red below).

```
"The shim was 6" high"
```

One possible solution is to use a different quote character around the constant. Any of the examples shown below will work.

```
'The shim was 6" high'
```

```
{The shim was 6" high}
```

```
"The shim was 6" high"
```

```
`The shim was 6" high`
```

Another solution is put two double quotes in a row (as highlighted dark blue in the example below). Panorama will convert these into a single quote and continue with the text constant.

```
"The shim was 6"" high"
```

Build in Constants: Pi, Carriage Return and Tab

Panorama has one built in numeric constant—**pi**. Use the Greek π symbol to access this value. For example the area of a circle can be calculated with this formula.

```
 $\pi$  * radius^2
```

To create the π symbol on the Macintosh press **Option-P**. On the PC, type **Alt-0254**.

Panorama has two built in text constants—**¶** (Carriage Return) and **↵** (Tab). For example three line address can be included in a formula like this.

```
"Suzette Elliot"+¶+892 Melody Lane"+¶+"Fullerton, CA 92831"
```

To create the **¶** symbol on the Macintosh press **Option-7**. On the PC, type **Alt-0182**.

To create the **↵** symbol on the Macintosh press **Option-L**. On the PC, type **Alt-0172**.

“Pipe” Delimited Constants

Panorama has a special **pipe** constant delimiter that is very handy for creating constants that have other types of quotes within the constant. The constant starts with a series of pipes, and doesn't end until an equal number of pipes. For example if the constant starts with 3 pipes it should also end with three pipes.

```
|||last="Elliot" first="Suzette" address="892 Melody Lane"|||
```

You can even embed pipes within a piped constant, like this:

```
|||| language=javascript code=|alert("Hello World");| |||
```

As you can see pipe delimited constants are very handy for creating text constants that contain computer code.

Fields

To use a field within a formula, type the name of the field into the formula. This formula adds up the sum of three fields.

```
SubTotal+Shipping+Tax
```

When a field is used in a formula it always refers to the value of that field in the current record in the current database (the database belonging to the topmost window). As you move from record to record the result of the computation will change depending on the values in that particular record. (The only exception to this rule is the **lookup(** and **grabdata(** functions, which may refer to fields in other records or even other databases.)

If a field name contains spaces, numbers, or punctuation marks in it, you must surround the name with chevron characters (« and »). (On the Macintosh press **Option-\<** to create the « chevron character and **Shift-Option-\<** to create the » chevron character. On Windows systems press **Alt-0171** to create the « chevron character and **Alt-0187** to create the » chevron character.) If the field name contains carriage returns, they must be represented with spaces. Here is a database with some unusual field names.

Price	Quantity	Zip Code	P/E Ratio
34.99	12	93221	15.67
12.99	154	50442	9.12
175.99	81	20165	45.83

The first two names can be used without chevrons, but the last two require chevrons because of spaces and punctuation in the names.

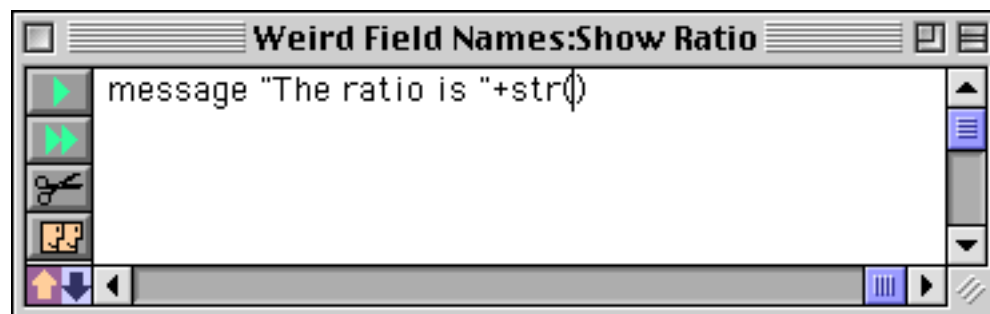
Price

Quantity

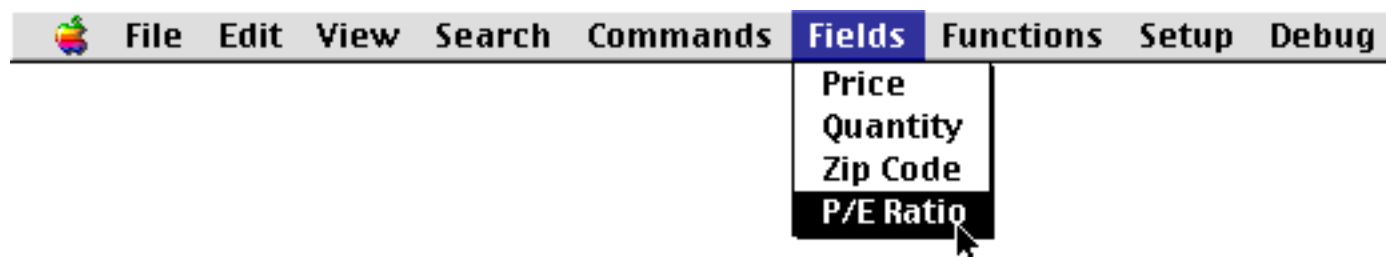
«Zip Code»

«P/E Ratio»

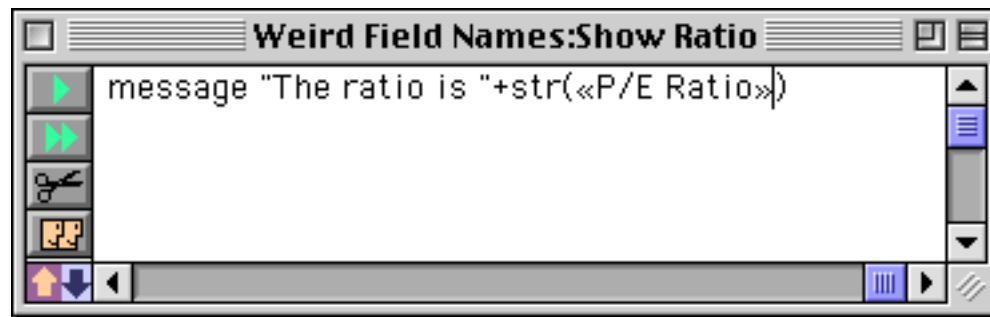
Formulas require field names to be spelled exactly as they appear in the database, with no typos allowed. Fortunately, Panorama can help you out with this. Start by positioning the insertion point where you want the field to appear.



Now pick the field from the **Field Menu**. This menu is available whenever you are editing a formula in a dialog, design sheet or procedure.



Panorama will type in the field name for you, including the chevrons if necessary (as they are in this case).



If the chevrons are not necessary (for example for [Price](#) or [Quantity](#)) Panorama will not include them.

Using the Current Field

A formula may use «» (see “[Special Characters](#)” on page 57) to refer to the current field without having to know what the current field is. For example, this formula converts the current cell to upper case.

```
upper («»)
```

If necessary, a formula can find out what the current field name is with the `info("fieldname")` function (see “[INFO\("FIELDNAME"\)](#)” on page 5375).

Line Item Fields

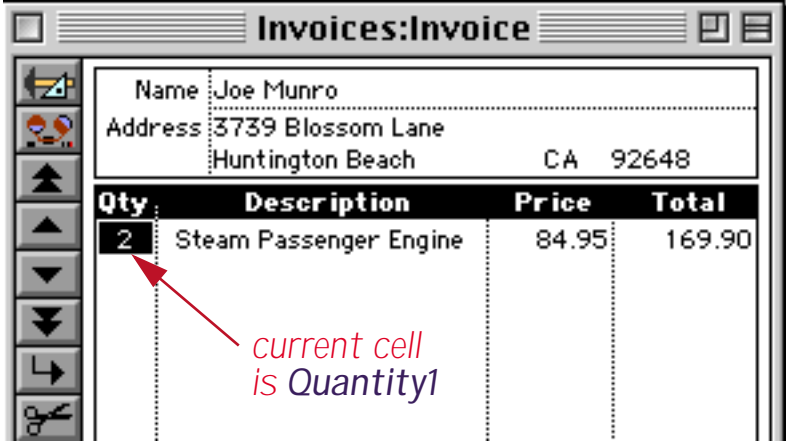
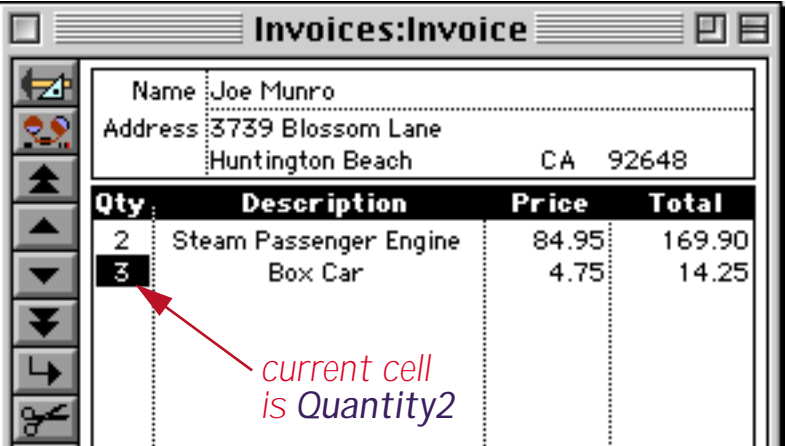
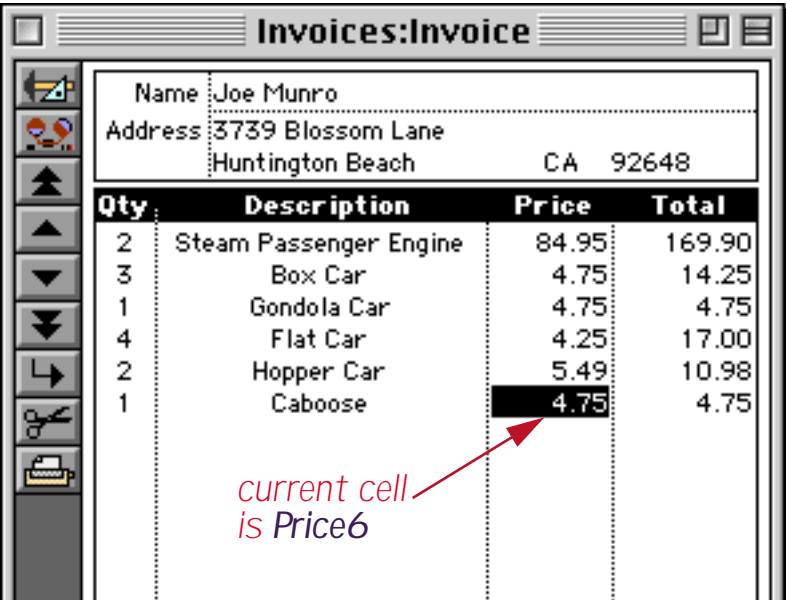
Line items are used for repeating items within a record (see “[Repeating Fields \(Line Items\)](#)” on page 222). Line item fields always end with a numeric suffix, for example [Qty1](#), [Qty2](#), [Qty3](#), etc. Line item fields can be used in formulas just like other fields, for example:

```
Quantity3*Price3
```

When the current cell is a line item field the Ω symbol (see “[Special Characters](#)” on page 57) can be used as an automatic numeric suffix. Panorama automatically adjusts the suffix depending on what cell is currently active. For example, consider this formula using the Ω symbol.

```
Quantity $\Omega$ *Price $\Omega$ 
```

When this formula is calculated, Panorama will automatically substitute the correct line item number for each Ω symbol as shown in this table.

Current Cell	Adjusted Formula
	$Quantity1*Price1$
	$Quantity2*Price2$
	$Quantity6*Price6$

If a formula containing the Ω character is used when the current cell is not a line item cell an error will occur.

To add up a series of line items you can use the `sum()` function. See “[Adding Line Item Fields](#)” on page 62.

Variables

A variable is a place in the computer where an item of data can be stored, kind of like a storage bin for a value. Variables may be created by procedures or by SuperObjects. Most procedures will use one or more variables to hold and transfer data as the program runs (see “[Variables](#)” on page 247 for more details on how variables can be created and used in procedures). Use a variable whenever you need to store a single data item so that you can use it later. Unlike a field, the value variable doesn’t change as you move from record to record, or, in the case of a global variable, even when you move from database to database.

Variable Names

Just as a house is identified by its address, a variable is identified by its name. A street address tells you exactly how to find a house or business. It doesn't tell you who or what is inside the house, however. Families may come and go, but the street address remains the same. In a similar way, a variable name identifies a place where data can be stored. The data may change, but the variable name remains the same.

Panorama allows any sequence of characters to be used as a variable name. However, if the variable name contains any punctuation (including spaces) it must be surrounded by the chevron characters « and ». (On the Macintosh press **Option-** to create the « chevron character and **Shift-Option-** to create the » chevron character. On Windows systems press **Alt-0171** to create the « chevron character and **Alt-0187** to create the » chevron character.) Here are some examples of typical variable names:

`x`

`birthDay`

`Counter`

`«Tax Rate»`

`«PrimeRate%»`

A variable name must be spelled exactly the same way every time, including upper and lower case. The variable name `birthDay` is not the same as `Birthday` or `birthday`. In fact, you could create three different variables using these three different names (although this is not recommended because it would be very confusing).

By the way, it's always ok to use chevrons around a variable name, even if the name doesn't have any punctuation. `«Counter»` is exactly the same as `Counter`, and they can be used interchangeably. So if you have any doubts about whether or not chevrons are necessary, go ahead and use them. No harm, no foul.

What's Inside A Variable?

By itself, a variable has no meaning, no value...until you put some data in it. When you use a variable in a formula or procedure, you are actually telling Panorama to use the contents of the variable.

A variable is sort of like a cup that you can pour anything into. A cup may contain water, soda, tea, or coffee. If you tell a person to drink the blue cup, what you really mean is to drink whatever liquid is in the blue cup. Each time they drink they may get a different liquid, depending on what the blue cup has been filled with.

Using a variable is similar. If you tell Panorama to calculate `X+Y` (where `X` and `Y` are variable names), what you really mean is "take whatever value is in `X` and whatever value is in `Y` and add them together."

It's important to remember that a variable name simply identifies the variable, but the name is not the variable itself. The name is like a placeholder for the real contents of the variable.

The Life Cycle of a Variable

A variable doesn't just appear by magic. It must be created, just as you have to build a house before you can move in. Once the variable has been created it can be used for storing a data item. However, variables don't last forever. Most variables eventually disappear without a trace. You can also force a variable to disappear at any time — see "[Destroying a Variable](#)" on page 249.

Panorama has five kinds of variables: local, window, fileglobal, global and permanent. The only difference between these three types of variables is how long they last before disappearing and when the variables are available.

Local variables are the most short-lived. A local variable disappears when the procedure that created the variable is finished. In addition, a local variable can only be used by the procedure that created it. If procedure A calls procedure B as a subroutine, procedure B cannot access the local variables created by procedure A. In fact, procedure B could create its own local variables with the same names as the local variables created by procedure A. Panorama keeps the local variables for each procedure completely separate from each other.

Window variables are associated with a particular window. A window variable is only accessible when the window it is associated with is on top, and the variable disappears completely when the window is closed. It is possible for several different windows to have window variables with the same name. In that case, each window variable may have a different value.

FileGlobal variables are associated with a particular database (file). A fileglobal variable is only accessible when the database it is associated with is the current database (on top), and the variable disappears completely when the file is closed. It is possible for several different files to have fileglobal variables with the same name. In that case, each fileglobal variable may have a different value. For many applications fileglobal variables are the best choice because there is no chance of an accidental conflict with a variable of the same name in another database.

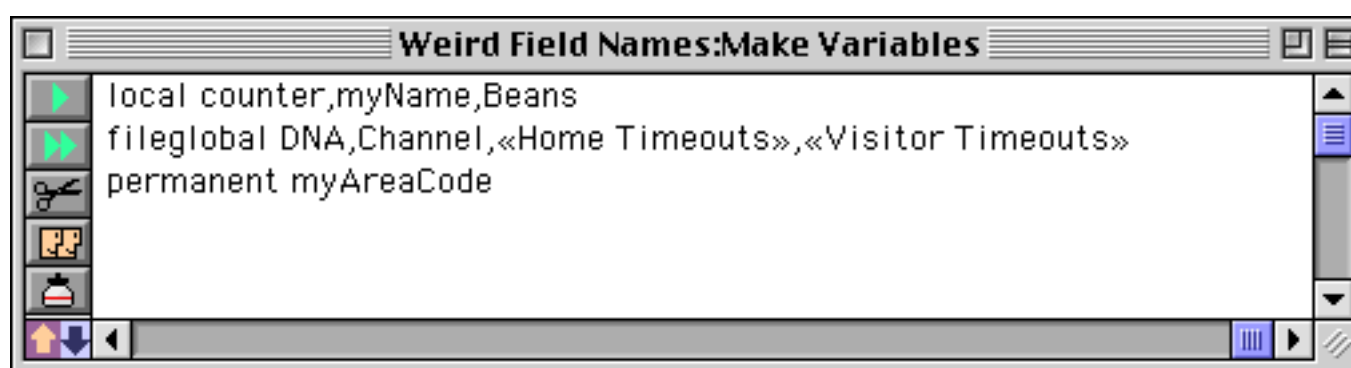
Global variables are relatively long-lived. A global variable doesn't disappear until you quit from Panorama. Even if you close the database, the global variable remains. Once a global variable has been created it can be accessed in any procedure, in any database or window, at any time. You should avoid using global variables unless you absolutely need universal access across databases for the value stored in the variable. If the value is only needed in one database it is much better to use a fileglobal variable to avoid the chance of an accidental conflict with another database using the same global variable name.

Permanent variables are almost immortal. When the database is saved, all permanent variables in that database are also saved. Like a fileglobal variable, a permanent variable is only accessible when the database it is created in is the current database, and a fileglobal variable disappears when you close the database. However, unlike a fileglobal variable, a permanent variable will re-appear like a phoenix from the ashes when you re-open the database. In fact, there are only two ways a permanent variable can permanently disappear. First, you can explicitly kill a permanent variable with the **unpermanent** statement. Secondly, you can create a permanent variable but never save the database.

Creating Variables in a Procedure

Panorama has five statements for creating variables in a procedure: **local**, **windowglobal**, **fileglobal**, **global**, and **permanent**. Each of these statements is exactly the same except for the type of variable created. The statement must be followed by a list of one or more variables to create, with each variable name separated from the next by a comma. (Remember: if a variable name contains punctuation, it must be surrounded by chevrons « and ».)

Here are a few examples of typical statements for creating variables:



There is no limit to the number of local, global, and permanent statements you use in your programs, and no limit to the total number of variables (except for scratch memory, see below).

Initializing Variables

Creating a variable creates a place to store data, but it doesn't actually put any data in the variable. It's kind of like a new house that no one has moved into yet. If you try to access the variable before any data has been put into it, an error occurs.

To put data into a variable, use an assignment statement. Here's the start of a procedure that creates a variable named `Count` and initializes it to zero. The variable is now ready to use.

```
local Count
Count=0
```

Sometimes you may not be sure if a global variable has been initialized yet. If it has not been, you want to initialize it. But if it has already been initialized, you don't want to disturb the value that is already there. You can get around this problem with the if error statement, as shown in this example.

```
global AreaCode
AreaCode=AreaCode
if error
    AreaCode="714"
endif
```

This procedure starts by creating a global variable named `AreaCode`. However, it's possible that `AreaCode` has already been created and initialized by another procedure. To test this, the procedure copies the variable to itself. If the variable is already initialized, there will be no error and the contents of the variable have not been disturbed. If the variable is brand new and has not been initialized, an error occurs. This error is trapped by the if error statement and the variable is initialized. If you have a number of variables that are always initialized as a group, you don't need to test each one. Just test one, and if an error occurs initialize the entire group of variables (see "[The Define Statement](#)" on page 244).

Variables and Data Types

A variable can hold any kind of data: text, numbers, and secondary data types like dates, times, points, rectangles, etc. In addition, you can change the type of data in a variable at any time. One minute the `AreaCode` variable can contain text, moments later it can contain a number. The variable takes on the data type of whatever data you copy into it.

SuperObject Variables

A number of Panorama SuperObjects™ have the option of linking to a variable or a field. These SuperObjects™ include the Text Editor, Data Button, Pop-up Menu, List, Sticky Button and Scroll Bar. If one of these objects is linked to a variable and the variable does not exist, Panorama will automatically create a global variable when it opens the form, and initialize the variable to empty text. Except for how it was created, this global variable is just like any other global variable and can be used freely in procedures and formulas.

Variable Name Conflicts

If two database files define a global variable with the same name, you've got a conflict. It's kind of like two families trying to share the same house. This can work if the two families have an arrangement, but if they don't the result is chaos.

The best solution to this problem is to avoid it. If you can, use a fileglobal variable instead of a global variable. If this is not possible, stay away from simple global variable names like `X`, `Payment`, `Count`, etc. If possible, choose names that incorporate the database name (or an abbreviation of the name), for example `InvoiceTaxRate`, `ReceivablesTotal`, or `APLastReconcileDate`.

Variable names (even for local variables) can also conflict with fields in a database. In this battle, the variable always wins. Panorama will use the data in the variable instead of the data in the field. Avoid variable names that are the same as field names.

Permanent Variable Tips

When the **permanent** statement creates a permanent variable, it really creates two variables: one in memory and one in the current database. The one in memory is an ordinary fileglobal variable. Whenever the database is saved, Panorama copies the contents of the fileglobal variable into the copy of the variable in the database itself, then saves the database. Just like any other data, the contents of the permanent variable are not saved unless the database itself is saved. However, if you have not made any other changes to the database, Panorama will not warn you if you attempt to close a database without saving changes to the permanent variable.

Whenever a database is opened, Panorama automatically creates fileglobal variables for any permanent variables associated with that database. Next it copies the values from the database into the fileglobal variables. The variables are now ready to use.

If you ever want to make a permanent variable un-permanent, use the **unpermanent** statement, which is followed by a list of variables you want to make unpermanent. This statement doesn't make the variables go away, but they will no longer be permanent. The **unpermanent** statement only affects variables that are permanent in the current database. The example below changes two permanent variables back into regular (non-permanent) fileglobal variables.

```
unpermanent myAreaCode,myZipCode
```

Special Characters

Formulas are very picky about special characters. You've got to use the right special character in the right spot—no substitutes are allowed.

For example, some people mistake the bracket [] characters for the parentheses (). On your keyboard, the parentheses are created by pressing **Shift** and the **9** or **0** keys. Another common mistake is using the \ (backslash) instead of the / (slash) for divide. The table below lists all the special characters used by formulas and shows how to type them.

Character	Name	Mac	PC
(left parenthesis	Shift-9	Shift-9
)	right parenthesis	Shift-0	Shift-0
[left bracket	[[
]	right bracket]]
{	left curly brace	Shift-[Shift-]
}	right curly brace	Shift-]	Shift-]
«	left chevron	Option-\	Alt-0171
»	right chevron	Shift-Option-\	Alt-0187
^	caret (raise to power)	Shift-6	Shift-6
*	asterisk (multiply)	Shift-8	Shift-8
÷	divide	Option-/	not available, use /
=	equal	=	=
≠	not equal	Option-=	not available, use <>
<	less than	<	<
>	greater than	>	>
≤	less than or equal	Option-<	not available, use <=
≥	greater than or equal	Option->	not available, use >=

Character	Name	Mac	PC
¶	paragraph	Option-7	Alt-0182
↵	export tab	Option-L	Alt-0172
§	section mark	Option-6	Alt-0167
¢	cents	Option-4	Alt-0162
‘	left smart quote	Option-]	Alt-0145
’	right smart quote	Shift-Option-]	Alt-0146
“	left smart double quote	Option-[Alt-0147
”	right smart double quote	Shift-Option-]	Alt-0148
Ω	omega (line items)	Option-Z	Alt-0166
π	pi	Option-P	Alt-0254

To use the **Alt** key on the PC you must hold down the **Alt** key, then press the numeric digits (for example **0182**) then release the **Alt** key. When you release the **Alt** key the special symbol will appear.

Working With Extremely Complex Formulas

Panorama has an internal 32000 byte buffer it uses for processing formulas. This allows very complicated formulas to be processed (since each function is represented in the buffer by a single byte, the actual formula can contain even more than 32,000 characters). However, it is possible to create a formula too large to fit into the buffer. When this happens Panorama generates an **Expression too complicated** error message.

To avoid this error Panorama allows you to create a larger expression buffer, letting you work with formula as complex as you want. To increase the size of the expression buffer, use the **formulabuffer** statement. This statement has one parameter, the number of bytes to make the formula buffer. For example, to make the formula buffer 120,000 bytes long, insert the following line into your procedure (you probably want to put this line into your **.Initialize** procedure, see “**.Initialize**” on page 382):

```
formulabuffer 120000
```

This would allow formulas up to six times as complicated as would be allowed normally.

The **formulabuffer** statement is semi-permanent: it applies to all formulas in all databases until you quit Panorama or change the setting again. If you want to cancel expanded buffer and go back to the internal buffer, use **formulabuffer** statement with a size of 0:

```
formulabuffer 0
```

The expanded formula buffer is not created until the procedure is run. That means that if the complex formula is in the same procedure as the **formulabuffer** statement, you won't be able to compile the procedure because the buffer hasn't been expanded yet. The best way to eliminate this problem is to put the **formulabuffer** statement into your **.Initialize** procedure.

How Large Should the Buffer Be?

Most users have never encountered the **Expression too complicated** error message and have no need to expand the buffer. If you do encounter this error, you should probably start by modestly expanding the buffer, perhaps to 40000 to 50000 bytes. If you still have a problem you can expand it further until the problem disappears.

However, if your database allows users to enter formulas that are out of your control (for example a formula that is automatically generated by selecting options on a form or a web page), you may wish expand the buffer in advance to a very large size, perhaps 100000 bytes.

Arithmetic Formulas

Panorama formulas are very adept at performing arithmetic—from simple addition to complex financial calculations. Arithmetic formulas usually work just like the ones you learned about in high school. Panorama has seven arithmetic operators, as shown in this table.

symbol	operator
+	add
-	subtract
*	multiply
/ or ÷	divide
^	raise to power
\	integer divide
mod	modulo (remainder)

The **^** operator (press **Shift-6**) raises the operand on the left to the power specified on the right. For example the formula

`2^3`

means raise 2 to the third power (equivalent to the mathematical formula 2^3).

The **** operator converts both operands into integers and then divides them. The result is also an integer. For example,

`19/5`

is 3.8 (a normal division), but

`19\5`

is 3. Notice that because this is an integer operation, the result is not rounded.

The **mod** operator computes the remainder after an integer division. For example the result of the formula

`19 mod 5`

is 4, but

`20 mod 5`

is zero. The result of the **mod** operator will always be an integer between zero and the value of the operand on the right (in this case 0, 1, 2, 3, or 4).

Dividing by Zero

Dividing by zero is, of course, a no-no. If you do attempt to divide by zero, Panorama will display an alert reminding you of this arithmetical impossibility. Sometimes, however, you may want to defy mathematical reality and divide by zero without getting slapped on the wrist. For example, since formulas treat empty data cells as zeros, attempting to divide by a cell that hasn't been entered yet will result in a divide by zero error. To bypass the error message, use the `divzero()` function instead of the `/` operator. The `divzero()` function returns zero if you attempt to divide by zero. For example, using the formula

```
Price/Qty
```

can result in a divide by zero error if `Qty` field is empty, but

```
divzero(Price,Qty)
```

will not.

Overflow/Underflow Problems

A number is a number, right? Well, not quite. You may remember that Panorama actually stores two different kinds of numbers—fixed digit and floating point, with fixed digit numbers being further divided into 0, 1, 2, 3, and 4 digit precision (see “[Numeric Data](#)” on page 249 of the *Panorama Handbook*). In a formula these differences may be important, since some numbers are too big or too small to be represented in some of the fixed point formats.

Formulas try to perform arithmetic using the final numeric type required for the answer. For example, if the result of a formula will be placed in a fixed 2 digit field, calculations will be performed in a fixed 2 digit format unless you force the formula to use another format. If the final destination is not a numeric field, arithmetic will be performed using floating point. Floating point is also used when the answer is not going to be stored in a field—for example formulas that are merged into auto-wrap text object (see “[Displaying Formulas in Auto-Wrap Text](#)” on page 602) or Text Display SuperObject (see “[Text Display SuperObjects™](#)” on page 608 of the *Panorama Handbook*).

Since the internal format used for arithmetic can vary depending on the final destination of the answer, the same formula can give different results depending on where it is used. For example, the formula

```
1/4
```

gives the result `0.25` if the result is a floating point field, but `0` if the result is a fixed 0 digit field.

A more subtle problem can occur if an intermediate calculation causes an overflow, underflow, or loss of precision. Often this can be fixed by re-arranging the formula. For example, this formula for computing sales tax can have problems if the result will be stored in a 2-digit fixed field.

```
total*taxrate/100
```

If the tax rate is 6.5%, the intermediate result of the division is `0.065`. But since 2-digit fixed point arithmetic is being used, this intermediate result will be rounded to `0.07`, resulting in an incorrect calculation. You can fix this formula by doing the multiplication first.

```
(total*taxrate)/100
```

You can also fix this formula by forcing all the numbers to floating point using the `float()` function.

```
float(total)*float(taxrate)/float(100)
```

If all the operands are in the same numeric format, the formula will calculate the result using that format, in this case floating point.

If you don't want to worry about overflow/underflow problems one solution is simply to make all numeric fields floating point. Floating point fields take up slightly more RAM than fixed point fields, but for most databases the difference isn't critical.

Adding Line Item Fields

Line items are used for repeating items within a record (see "[Repeating Fields \(Line Items\)](#)" on page 222 of the *Panorama Handbook*). Line item fields always end with a numeric suffix, for example Qty1, Qty2, Qty3, etc. Line items can be added up just like ordinary fields:

```
Qty1+Qty2+Qty3+Qty4+Qty5
```

You can also use the `sum(` function to add up line item fields:

```
sum("QtyΩ")
```

Using the `sum(` function is easier to type, and it is slightly faster than regular addition when used in the design sheet or a procedure. (Ordinary addition is faster than the `sum(` function when used in a **Formula Fill**.)

When you use the sum function, don't forget to include the quotes around the field name as shown above, and don't forget the Ω symbol (see "[Special Characters](#)" on page 57). To learn how to perform calculations within line item fields see "[Line Item Fields](#)" on page 52.

Warning: The `sum(` function is not compatible with the Design Sheet's **Spreadsheet Mode** (see "[Spreadsheet Mode Calculations](#)" on page 303 of the *Panorama Handbook*). If you are using Spreadsheet Mode you must add up the items field by field (i.e. Qty1+Qty2+Qty3...).

Basic Numeric Functions

These functions perform various mathematical operations. Each of these functions takes one or more numeric parameters and returns a numeric result.

Function	Reference Page	Description
abs(number)	Page 5010	This function returns the absolute (positive) value of the numeric parameter. In other words, negative numbers are converted to positive numbers while positive numbers remain positive.
divzero(numerator,denominator)	Page 5178	This function divides two numbers. However, unlike the / operator, the divzero(function does not care if you attempt to divide by zero. If you attempt to divide by zero, this function simply returns zero.
fix(number)	Page 5254	This function truncates a number to an integer. It always truncates towards zero. For example <code>fix(-4.6)</code> is -4, while <code>int(-4.6)</code> is -5. For positive numbers the int(and fix(functions are identical. Don't confuse this function with the fixed(function, which converts numbers from floating to fixed point format.
fixed(number)	Page 5255	This function forces a number to fixed point format, using the least number of digits possible. Since formulas usually perform this conversion automatically, you probably won't ever need this function. Don't confuse this function with the fix(function, which truncates a number to an integer but does not change the type of the data.
float(number)	Page 5256	This function forces a number to a floating point format. You may need to use floating point to get around overflow, underflow, and accuracy problems that can occur when using fixed point arithmetic.
int(number)	Page 5458	This function truncates a number to an integer. It always truncates towards negative infinity. For example <code>int(-4.6)</code> is -5, while <code>fix(-4.6)</code> is -4. For positive numbers the int(and fix(functions are identical.

Function	Reference Page	Description
max(number,number)	Page 5523	This function compares two numbers and returns the larger value. If you need to compare more than two numbers, you can nest this function within itself, for example <code>max(a,max(b,c))</code> .
min(number,number)	Page 5530	This function compares two numbers and returns the smaller value. If you need to compare more than two numbers, you can nest this function within itself, for example <code>min(a,min(b,c))</code> .
numsandwich(value,extra)		This function is similar to the sandwich function, but for numbers. If the value is zero the result is zero, but if the value is not zero the result is the value plus the extra. For example, this could be useful for calculating the size of an object plus a border. If the object size is zero, the border is omitted also, for example <code>numsandwich(20,7)</code> will be equal to <code>27</code> , but <code>numsandwich(0,7)</code> will be equal to <code>zero</code> .
randominteger(startnum,endnum)		Returns a random integer value greater than or equal to the startnumber and less than or equal to the end number.
round(number,step)	Page 5682	This function rounds a number to the nearest step. You can use any value you want for the step: <code>1</code> , <code>10</code> , <code>0.5</code> , whatever. For example, you could use the formula <code>round(Quantity,12)</code> to round the quantity to the nearest dozen. The quantity <code>16</code> will be rounded to <code>12</code> ; the quantity <code>20</code> will be rounded to <code>24</code> . Since dates are treated as numbers (see “ HTML Generating Functions ” on page 105) you can use this function round to the nearest week. Use a step value of <code>7</code> (7 days per week), for example <code>round(Date,7)</code> .
rnd()	Page 5681	This function returns a random number between 0 and 1. Each time you use this function it will return a different number. If you need a random number in a different range just adjust the output of this function. For example, to get a random number between 1 and 10, use the formula <code>int(1+10*rnd())</code> . Notice that even though this function has no parameters, you must still include the empty parentheses after the function name.
sum("lineitemΩ")	Page 5816	This function adds up all the instances of a line item field within the current record. You must specify the name of the line item field followed by the Ω character (see “ Special Characters ” on page 57). The whole thing must be surrounded by quotes, for example <code>sum("QtyΩ")</code> . This example is the same as the formula <code>Qty1+Qty2+Qty3...</code> but much easier to type! Warning: The <code>sum(</code> function is not compatible with the Design Sheet’s Spreadsheet Mode (see “ Spreadsheet Mode Calculations ” on page 303 of the <i>Panorama Handbook</i>). If you are using Spreadsheet mode you must add up the items field by field (i.e <code>Qty1+Qty2+Qty3...</code>).
zeroblank(number)	Page 5915	This function tells Panorama to store zero as an empty space. If the final formula result is not zero, this function has no effect. The <code>zeroblank(</code> function is handy when you want to leave the result of a calculation blank if one of the operands are blank. For example, if you use the formula <code>zeroblank(Qty*Price)</code> , the result will be empty if either the quantity or price is empty.

Scientific Functions

These functions perform various log, trig, and exponential calculations. Each of these functions takes one or more numeric parameters and returns a numeric result.

The trig functions listed in this table normally use radians to measure angles (1 radian = $180/\pi$ degrees). In a procedure the `degree` statement may be used to temporarily switch Panorama's trig functions to use degrees instead of radians (see "[DEGREE](#)" on page 5155 of the *Panorama Reference*). The `radians` statement switches the mode back to radians (Panorama also switches back automatically when the procedure is finished). For example, the procedure below calculates the tangent of 30 degrees, not 30 radians.

```
degree
height=tan(30)
```

Calculations performed outside of a procedure always use radians (for example in a Text Display SuperObject). If you need to convert degrees into radians you can simply multiple the number of degrees by $180/\pi$ (see "[Special Characters](#)" on page 57), for example `tan(30*180/π)`.

Function	Reference Page	Description
<code>arccos(number)</code>	Page 5030	This function calculates the inverse cosine of a number. The number must be between -1 and +1. The result is normally in radians, but may be in degrees if the degree statement has been used (see " DEGREE " on page 5155 of the <i>Panorama Reference</i>).
<code>arccosh(number)</code>	Page 5031	This function calculates the inverse hyperbolic cosine of a number. The number must be between 1 and ∞ .
<code>arcsin(number)</code>	Page 5032	This function calculates the inverse sine of a number. The number must be between -1 and +1.
<code>arcsinh(number)</code>	Page 5033	This function calculates the inverse hyperbolic sine of a number.
<code>arctan(number)</code>	Page 5034	This function calculates the inverse tangent of a number. The result is normally in radians, but may be in degrees if the degree statement has been used (see " DEGREE " on page 5155 of the <i>Panorama Reference</i>).
<code>arctanh(number)</code>	Page 5035	This function calculates the inverse hyperbolic tangent of a number. The number must be between -1 and +1.
<code>cos(number)</code>	Page 5128	This function calculates the cosine of an angle. The angle is normally specified in radians, not degrees. To convert degrees to radians, divide by $180/\pi$, which is 57.2958. For example <code>cos(A*180/π)</code> calculates the cosine of A, where A is in degrees. It is also possible to modify the action of Panorama to use degrees instead of radians for all trig functions, see " DEGREE " on page 5155 of the <i>Panorama Reference</i> .
<code>cosh(number)</code>	Page 5130	This function calculates the hyperbolic cosine of a number. The result will be a value between 1 and ∞ .
<code>exp(number)</code>	Page 5202	This function raises e to a number. For example, the formula <code>exp(10.2)</code> is equivalent to $e^{10.2}$. Incidentally, e is a constant that is used in many mathematical formulas. Its approximate value is 2.71828.
<code>fact(number)</code>	Page 5214	This function calculates the factorial of a number. For example, the formula <code>fact(4)</code> is equivalent to $4!$ or $4*3*2*1$. You can calculate the factorial of any integer from 0 to 170.
<code>log(number)</code>	Page 5492	This function calculates the natural logarithm (base e) of a number.
<code>log10(number)</code>	Page 5493	This function calculates the common logarithm (base 10) of a number.

Function	Reference Page	Description
sin(angle)	Page 5772	This function calculates the sine of an angle. The angle is normally specified in radians, not degrees. To convert degrees to radians, divide by $180/\pi$, which is 57.2958. For example $\sin(A*180/\pi)$ calculates the sine of A, where A is in degrees. It is also possible to modify the action of Panorama to use degrees instead of radians for all trig functions, see “ DEGREE ” on page 5155 of the <i>Panorama Reference</i> .
sinh(angle)	Page 5774	This function calculates the hyperbolic sine of a number.
sqr(angle)	Page 5791	This function returns the square root of the number.
tan(angle)	Page 5841	This function calculates the tangent of an angle. The angle is normally specified in radians, not degrees. To convert degrees to radians, divide by $180/\pi$, which is 57.2958. For example $\tan(A*180/\pi)$ calculates the tangent of A, where A is in degrees. (Note: The tangent of $\pi/2$ (90°) is ∞ , which results in an overflow error.) It is also possible to modify the action of Panorama to use degrees instead of radians for all trig functions, see “ DEGREE ” on page 5155 of the <i>Panorama Reference</i> .
tanh(number)	Page 5843	This function calculates the hyperbolic tangent of a number. The result will be a value between -1 and +1.

Financial Functions

These functions calculate financial data, including loan payments, future value, and present value. They are designed to be compatible with the same functions in Microsoft Excel®. The financial functions are based on the following formula.

$$pv(1+rate)^{periods} + payment(1+rate \times begin) \times ((1+rate)^{periods} - 1) / rate + fv = 0$$

Function	Reference Page	Description
<code>pmt(rate,periods,amount,fv,begin)</code>	Page 5604	<p>This function calculates the periodic payment required to pay off a loan. The rate is the interest rate of the loan per period. Periods is the term of the loan expressed in payment periods, for example 36 months for a three year loan that is paid monthly. Amount is the amount being borrowed. The fv (future value) and begin values are optional, and should usually be set to zero.</p> <p>For example, suppose you are taking out a 36 month loan of \$20,000 to buy a car. If the annual interest rate is 13.5% (1.125% compounded monthly), what would the monthly payment be?</p> <pre>pmt(0.135/12 , 36 , 20000 , 0 , 0)</pre> <p>The monthly payment is \$678.71.</p>
<code>fv(rate,periods,payment,pv,begin)</code>	Page 5283	<p>This function calculates the future value of an investment. Rate is the interest rate per period. Periods is the term of the investment, for example ten years or 48 months. The pv is the present value of the investment, for example the starting balance in a savings account. Begin should be either 1 or 0; 1 if the payments occur at the beginning of the period, 0 if the payments occur at the end of the period.</p> <p>For example, to calculate the final balance in a savings plan when you invest \$500 per year for 10 years at 9% annual interest use the formula—</p> <pre>fv(0.09 , 10 , -500 , 0 , 1)</pre> <p>At the end of ten years you would have \$8280.15. What if this savings plan already has \$2000 in it at the time you start this 10 year savings program? The new formula would be—</p> <pre>fv(0.09 , 10 , -500 , -2000 , 1)</pre> <p>At the end of 10 years you would have \$13,014.87.</p>
<code>pv(rate,periods,payment,fv,begin)</code>	Page 5621	<p>This function calculates the present value of an investment. Rate is the discount rate, periods is the periodic investment, and payment is the periodic payment. The fv is an optional lump sum at the end of the final period; use zero if there is no lump sum. Begin specifies whether payments are received at the beginning or end of each period—1 for beginning or 0 for end.</p> <p>Present value is a variation of the old theme that a bird in the hand is worth two...well, you know. It's better to get \$1000 now instead of \$1000 next year, but how much better? The present value computation puts a numeric value on time and money.</p> <p>For example, suppose you find an investment opportunity that promises to pay you \$1,000 per year for the next 3 years. Assuming the current interest rate is 10% per year, how much are these payments worth right now?</p> <pre>pv(0.1 , 3 , 1000 , 0 , 0)</pre> <p>The computation shows that \$3000 paid over 3 years is worth \$2486 right now (assuming 10% interest).</p>

Text Formulas

Formulas can work on text as well as numbers. Formulas can combine two or more pieces of text, extract a portion of a piece of text (for example the area code or last name), or even re-arrange the text. Formulas can also convert numbers into text and back again.

Programmers call a piece of text a **string**, referring to the fact that the text is made up of a string of characters. Since this is such a handy term we'll use it ourselves. So whenever you see the word **string** think "piece of text."

Where do strings come from? Most strings come from the database itself. Any text or choice field can be used as a string. You can also store strings in a variable (see "[Variables](#)" on page 53), or put a string right into the formula itself (see "[Constants](#)" on page 49).

Gluing Strings Together

The simplest operation that can be performed on two strings is sticking them together, also called **concatenation**. To glue strings together use the + operator. This operator attaches the string on the right to the end of the string on the left. For example the formula

```
"abc"+"def"
```

produces the result **abcdef**. To attach the word **Mr.** to the beginning of a last name field use the formula

```
"Mr. "+«Last Name»
```

(Of course, you better be sure everyone in the database is a man!).

You can use more than one + operator to stick several strings together at once. For example to combine separate first and last names into a single string using the format **Last, First** use this formula:

```
«Last Name»+", "+«First Name»
```

Another way to glue strings together is with the **sandwich()** function ([reference page 5689](#)). This function combines up to three items of text: a **prefix**, a **suffix**, and the **root** text. The **prefix** and **suffix** are slapped on the ends of the **root**, just like a sandwich. However, if the **root** is empty (sort of like a sandwich with no meat!) the **prefix** and **suffix** are also left off, just as you wouldn't bother to make a sandwich without any meat.

Let's revisit our previous example with the **sandwich()** function. The previous formula will work fine as long as there is a first name. But if the first name is empty, the formula will produce an extra comma, for example **Jones, .** The sandwich function can solve this problem:

```
«Last Name»+sandwich(", ",«First Name»,")
```

If the **First Name** field contains a name, the **sandwich()** function will slap the prefix in front of the name (in this case the prefix is a comma and a space). But if the **First Name** field is empty, the sandwich() function will also leave off the prefix. All the formula will produce is the **Last Name**, with no extra comma and space.

The **rep()** function ([reference page 5662](#)) repeats an item of text by concatenating it to itself over and over. The number of times the item is repeated is specified by the second parameter, which must be an integer. For example, this formula will create twenty asterisks in a row:

```
rep(" ",20)
```

This is exactly the same as the formula:

```
"*****"
```

The `rep()` function, however, is less prone to error, and the count can be changed easily or even vary dynamically. Here is a function which adds leading asterisks to a number so that there are always 15 characters.

```
rep(" ",15-length(pattern(Amount,"$#,##")))+pattern(Amount," $#,##")
```

This formula is perfect for displaying numbers with an auto-wrap text object or Text Display SuperObject. The numbers will be padded with asterisks, for example ***** \$4,983.45.

Functions for Taking Strings Apart

These functions return portions of a string. See also “[Taking Strings Apart \(Text Funnels\)](#)” on page 69, “[String Modification Functions](#)” on page 80, “[Text Arrays](#)” on page 93 and “[HTML Tag and Tag Parsing Functions](#)” on page 101.

Function	Reference Page	Description
<code>after(text,tag)</code>		This function extracts all of text after a specified tag (sequence of characters). If the tag doesn't exist within the text the function returns "".
<code>before(text,tag)</code>		This function extracts all of text before a specified tag (sequence of characters). If the tag doesn't exist within the text the function returns "".
<code>firstline(string)</code>		This function extracts the first line from the text.
<code>firstword(string)</code>		This function extracts the first word from the text (the text up to the first space).
<code>lastline(string)</code>		This function extracts the last line from the text.
<code>lastword(string)</code>		This function extracts the last word from the text (the text from the last space to the end).
<code>left(string,len)</code>		Extracts characters from the left edge of the text. For example <code>left(text,2)</code> extracts the leftmost two characters.
<code>mid(string,len)</code>		Extracts characters from the middle of the text. For example <code>mid(text,6,4)</code> extracts four characters starting with the sixth character.
<code>nthline(string,num)</code>		This function extracts the nth line from the text. For example <code>nthline(text,4)</code> extracts fourth line.
<code>nthword(string,num)</code>		This function extracts the nth word from the text. For example <code>nthword(text,7)</code> extracts seventh word.
<code>removeprefix(text,prefix)</code>		This function checks to see if a text item starts with a prefix. If it does, the prefix is removed.
<code>removesuffix(text,suffix)</code>		This function checks to see if a text item starts with a suffix. If it does, the suffix is removed.

Function	Reference Page	Description
<code>right(string,len)</code>		Extracts characters from the right edge of the text. For example <code>right(text,7)</code> extracts the rightmost seven characters from the text.
<code>snip(string,startposition,count)</code>		This function removes (snips!) one or more characters from the middle of an item of text. The startposition specifies the first character removed, the count is the number of characters to remove. (Note: This function requires the startposition to be a positive number.) If count is -1 then all the text from the start position to the end of the text is snipped, otherwise the count must be a positive number.
<code>textafter(string,tag)</code>		This function extracts the text after the tag. The tag may be one or more characters long. If the tag doesn't occur in the text then the entire original string is returned. For example <code>textafter("someone@isp.net","@")</code> will return <code>isp.net</code> .
<code>textbefore(string,tag)</code>		This function extracts the text before the tag. The tag may be one or more characters long. For example <code>textbefore("someone@isp.net","@")</code> will return <code>someone</code> . If the tag doesn't occur in the text then the entire original string is returned.
<code>trim(string,len)</code>		This function removes characters from the right edge of the text. For example <code>trim(text,4)</code> removes the last four characters from the text.
<code>trimleft(string,len)</code>		This function removes characters from the left edge of the text. For example <code>trimleft(text,2)</code> removes the first two characters from the text.

Taking Strings Apart (Text Funnels)

Sometimes you may have an item of text where you only need a portion of the text and want to strip off the beginning and or the end of the text. In addition to the functions in the previous section Panorama has a special tool for stripping off the ends of a text item. This tool is called a **text funnel**. Text funnels are powerful tools, however, many users find them a bit difficult to figure out. In recent years we've added many functions that can perform most of the operations that a text tool can perform. Before deciding to use a text funnel you may want to check out "[Functions for Taking Strings Apart](#)" on page 68, "[String Modification Functions](#)" on page 80, "[Text Arrays](#)" on page 93 and "[HTML Tag and Tag Parsing Functions](#)" on page 101.

A text funnel is used a bit differently than other Panorama functions and operators. The text funnel always follows the text item that is being "stripped." In a sense a text funnel has three parameters, the text item, start, and end. But as you can see below, these parameters are arranged quite differently than they are for other functions:

```
<text item>[<start>,<end>]
```

The first parameter, **text item**, is the item of text which will be stripped to get the final result. This may be a field, a variable, or an entire formula (as long as it produces a text item as its final result). If you use an entire formula you should put parentheses around the formula.

The second parameter, **start**, specifies the first character you want to include in the final output. For example if you want to strip off the first three characters the start should be 4 (because the 4th character is the first one we want to keep). If the starting position is past the end of the text all the text will be stripped out and the formula is left with an empty text item.

The third parameter, **end**, specifies the last character you want to include. For example, if you want to strip off everything after the 12th character, the end should be 12. If the starting position is after the ending position, all the text will be stripped and the formula is left with an empty text item.

The real trick in setting up text funnels is deciding what the start and end parameters should be. The following sections will describe several techniques for setting up these parameters.

Numeric Start and End Positions

The simplest way to specify starting and ending positions is with a number. Positive numbers are counted from the beginning of the original text item (1 is the first character in the original text item). Negative numbers are counted from the last character of the original text item (-1 is the last character).

Our first example removes the first character from the **Notes** field.

```
Notes[2,-1]
```

The next example does the exact opposite—it removes the last character from the **Notes** field.

```
Notes[1,-2]
```

By using the same number for the start and end a text funnel can strip out a single character. The procedure below uses the text funnel **[1,1]** to check to see if the first character of the phone number is a (. If so, it uses another text funnel to strip out the area code.

```
if Phone[1,1]="("
  AreaCode=Phone[2,4]
endif
```

A procedure can use a variable to pre-load the start and end positions. The procedure below will strip out everything starting with the phrase **Private Notes Below ---**.

```
local X
X=search(Notes,"Private Notes Below ---")
if X#0
  PublicNotes=Notes[1,X-1]
else
  PublicNotes=Notes
endif
```

Specifying Numeric Length Instead of Position

An alternate form of text funnel allows you to specify the length of the text to be stripped out, instead of the ending position. This alternate form simply uses a semicolon instead of a comma:

```
<text item>[<start>;<length>]
```

The **length** specifies the number of characters from the starting position. A positive length means that the stripped text begins at the starting position and extends to the right. A negative length means that the stripped text begins at the starting position and extends to the left. The character at the starting position is always included (unless the length is zero).

Let's look at two examples of this technique. The first extracts the area code from a long distance phone number.

```
Phone[2;3]
```

The next example strips out the local phone number (the last 8 characters).

```
Phone[-1;-8]
```

If the original text item is too short to fulfill the request the text funnel will take whatever it can get. For example, if the phone number is only 3 characters long, the value in **LocalNumber** will be 3 characters long.

Start/End Positions by Character Matching

The previous section described how to strip out text by absolute numeric position within the original text (for example from character 3 to character 8). Another technique is to specify not the absolute position, but the character value where stripping should begin and/or end. For example instead of telling the text funnel to strip off everything before position 5, you tell the funnel to strip off everything before the first \$ character, or everything after the last % character. The text funnel scans the original text looking for a matching character, and then strips the text accordingly.

To specify a starting or ending position by character matching, simply supply a character instead of a number. For example, suppose you had a field named `Line` that contained data like this:

```
X2245A Tape Cartridge $22.95
```

To extract just the price from `Line` you could use this text funnel.

```
Line["$",-1]
```

This formula will take `Line` and strip off everything in front of the first dollar sign. In our example this will be the value `$22.95`. If there is no dollar sign in `Line` then the result will be empty text (`""`).

Notice that the output of a text funnel is always a text item, not an actual number. If you wanted to convert this to a number you would have to remove the dollar sign with an additional text funnel and use the `val()` function ([reference page 5886](#)).

A text funnel can use a character value for either the starting or ending position, or both. Here is an example that extracts the hour from the time by stripping off everything after the first colon:

```
Time[1,":"]
```

Both of these examples are developed further in the next section.

Cascading Text Funnels

The examples in the previous section both have a problem: they don't strip off enough text. The first example strips off the price but leaves the dollar sign (for example `$45.67`). The second example strips the hour from the time but leaves the colon (for example `9:`). These problems can be solved by using two text funnels in a row.

Adding a second text funnel is easy—just enter it after the first funnel. This example strips off the \$ symbol from the beginning of the price using a regular numeric position text funnel.

```
Line["$",-1][2,-1]
```

The table below shows how some typical data would be processed by this formula.

Original Data	After ["\$",1]	After [2,-1]
X2245A Tape Cartridge \$22.95	\$22.95	22.95
AF8899 Data Cassette \$7.80	\$7.80	7.80
XB3 Head Cleaner \$19.50	\$9.50	9.50

This example strips off the hour from the time—including the pesky extra colon.

```
Time[1,":"][1,-2]
```

Once again, the table shows how the data is processed by each text funnel.

Original Data	After [1,":"]	After [1,-2]
9:42 AM	9:	9
3:07:12 PM	3:	3
11:23 AM	11:	11

These examples show two text funnels cascaded together, but there is no limit to the number of text funnels you can use in a row. Each funnel chops away at the text until you have just the text you want. Usually the best approach to developing a series of cascaded funnels is to develop one funnel at a time. Make sure one funnel really does what you want and expect it to before adding the next one.

Character Matching in Reverse Gear

If the character to be matched is preceded by a minus sign the text funnel will match the last instance of the value in the original text instead of matching the first.

The example below strips out the year from an appointment. The formula assumes that there is a date in the format `mm/dd/yy` somewhere in the text item. The funnel will attempt to match up the last `/` symbol in the original text.

```
Year="19"+Appointment["-/", -1][2;2]
```

Here is how the data is processed.



Original Data	After ["-/", -1]	After [2;2]
Lunch with Bob 3/4/01	/01	01
call Joan 4/2/99 3PM	/99 2PM	99
10/7 L's Birthday	/7 L's Birthday	7
call Ted	call Ted	al

The last two lines above shows the hazards of making faulty assumptions. Neither line contains a valid `mm/dd/yy` date. The result in this case is a bogus year. Unfortunately there is no magic pill fix for this kind of problem. As a programmer you must think of, check for and process every possible option. If you absolutely know that there will be a date in the text item, fine. If not, you'll have to write a more complicated procedure to check for a properly formatted date before you strip out the year. Here's an example of a more robust procedure.

```
if Appointment notmatch "*/*/*"
message "Sorry, the appointment has no year!"
  stop
endif
Year="19"+Appointment["-/", -1][2;2]
```

This procedure could still be fooled—for example data containing two dates would trip it up. Designing a completely foolproof procedure is left as an exercise to the reader.




Stripping Out Individual Words

One of the most common needs is to strip out a single word at the beginning, middle or end of a text item. This is easily done by using a space as the matching character value. You'll need to look at the formulas in this section very carefully. Don't confuse a space (" ") with an empty text item (""). They're not the same thing. (For clarity, the samples below showing how the data is processed use  to show a space whenever it is at the beginning or end of a text item. For example, now means space followed by now.)

Here's a formula that extracts the first word from an item of text by stripping off all the rest of the words.

```
Original[1," "][1,-2]
```




Here's how this formula would process several sample text strings.

Original Data	After [1," "]	After [1,-2]
Now is the time	Now 	Now
Boston reports 23 degrees	Boston 	Boston
Apple stock up 5 points	Apple 	Apple

This next formula does the exact opposite: it strips off the first word, leaving the rest of the words.

```
Original["",-1][2,-1]
```







Here is how this formula would process the same sample strings as before.

Original Data	After [1," "]	After [1,-2]
Now is the time	 is the time	is the time
Boston reports 23 degrees	 reports 23 degrees	reports 23 degrees
Apple stock up 5 points	 stock up 5 points	stock up 5 points

You can cascade these two text funnels to produce a formula that extracts the 2nd word from the original text, stripping off the rest.

```
Original["",-1][2,-1][1," "][1,-2]
```

Here is how this formula would process the same sample strings as before.




Original Data	After ["",-1]	After [2,-1]	After [1," "]	After [1,-2]
Now is the time	 is the time	is the time	is 	is
Boston reports 23 degrees	 reports 23 degrees	reports 23 degrees	reports 	reports
Apple stock up 5 points	 stock up 5 points	stock up 5 points	up 	up

This process can be repeated indefinitely. However, a better approach is probably to use the [array\(\)](#) function with space as a separator character. See [reference page 5036](#) to learn more about this function.

Here's a simple formula that extracts the last word from a text item, stripping off the earlier words (if any).

```
Original["-",-1][2,-1]
```

Here's how this formula would process several sample text strings.

Original Data	After ["- ",-1]	After [2,-1]
Now is the time	 time	time
Boston reports 23 degrees	 degrees	degrees
Apple stock up 5 points	 points	points

A close examination will show that this is exactly the same as the first example but with an extra minus sign to specify the last space instead of the first space.

Multiple Matching Characters for Start/End Position

Sometimes you may need to use multiple character values to specify the starting or ending position of a text funnel. Any one of these character values will match up with the original text. For example, a sentence may end with a period, a question mark, or an exclamation mark. To use more than one matching character simply list each character separated by commas. Here is an example that extracts the first sentence from a letter. All the text after the first sentence is stripped off.

```
Letter[1,".,?,!"]
```

You can include a comma as one of the character values. This example extracts everything up to the first semicolon, comma, or colon. All the text after that point is stripped off.

```
Description[1,";,,, :"]
```

If you use alphabetic values, don't forget that upper and lower case are separate values, even for the same letter. This example extracts **am** or **pm** from a text item.

```
Appointment["a,p,A,P";2]
```

It's also possible to specify a range of matching characters, for example **0** through **9** or **A** through **Z**. To specify a range the starting and ending characters must be separated by a dash, for example **"0-9"**. The range will include all characters between the two characters on the ASCII table (see "[Characters and ASCII Values](#)" on page 87.)

Here is an example that extracts the frequency from a radio station. The call letters are stripped off.

```
Station["0-9",-1]
```

Here's how this formula would process several sample text strings.

Original Data	After ["0-9",-1]
KFI 640 AM	640 AM
KLSX 97.1 FM	97.1 FM
KFAC 105.1 FM	105.1 FM
KROQ 106.7 FM	106.7 FM

A text funnel can combine multiple character ranges, or combine a range with one or more separate character values. The next example strips off everything before the first number, or before the first dollar sign (whichever comes first).

```
Line["0-9,$",-1]
```

Here's how this formula would process several sample text strings.

Original Data	After ["0-9,\$",-1]
Tape Cartridge \$22.95	\$22.95
Data Cassette 7.80	7.80
XB3 Cleaner \$19.50	3 Cleaner \$19.50

The last line shows a possible pitfall of this text funnel. Text funnels rely on consistent patterns in the data. If there isn't a pattern you can identify accurately, you won't be able to design a funnel to strip the text apart reliably. In this case a more reliable pattern would be to notice that the price is always the last word of **Line**, so the text funnel below will strip off the price reliably.

```
Line["- ",-1][2,-1]
```

Be sure to test your text funnels with a wide variety of sample data to make sure you have identified a consistent pattern.

As mentioned in the previous section, putting a minus sign in front of the character value tells the text funnel to find the last matching character, instead of the first. This works for character ranges too. This example extracts the item name from **Line** by stripping off everything after the last letter.

```
Line[1,"-A-Z,a-z"]
```

Here's how this formula would process our sample text strings.

Original Data	After [1,"-A-Z,a-z"]
Tape Cartridge \$22.95	Tape Cartridge
Data Cassette 7.80	Data Casette
XB3 Cleaner \$19.50	XB3 Cleaner

Although this example has two ranges (A-Z or a-z) only one minus sign is needed (at the beginning). If the first character is a minus sign, the text funnel will always look for the last matching character in the original text.

Non-Matching Character for Start/End Position

The previous examples have all used one or more characters that must match a character in the original text item. By using the \neq symbol (see "[Special Characters](#)" on page 57) you can specify that the text funnel should begin (or end) with the first character (or characters) that does not match. For example, you might want to match with the first character that is not a number, or the last character that is not a space.

An example should make this clearer. Suppose you have imported some numbers that have one or more asterisks in front of them, and you want to strip off the asterisks. The text funnel in this formula will set the starting position to the first character in the original text that is not an asterisk.


```
Imported["≠*", -1]
```




Here's how this formula would process some sample text strings.

Original Data	After ["≠*", -1]
****23.67	23.67
***782.12	782.12
*****2.98	2.98

You can use this feature to strip off leading spaces.

```
Name["≠ ", -1]
```

Here's how this formula would process some sample text strings (leading spaces are shown as  for clarity).

Original Data	After ["≠ ", -1]
  Jeff Nance	Jeff Nance
 Williams	Williams

(An easier way to strip leading and trailing spaces is to use the `strip()` function, which is designed for that purpose. See [reference page 5801](#) for more information about this function.)

The example below specifies that the starting position should be the first character that is not a letter, not a comma, and not a space. It extracts the zip code or Canadian postal code from an address.

```
CityStateZip["≠A-Z,a-z,,, ", -1]
```

Here's how this formula would process some typical addresses.

Original Data	After ["≠A-Z,a-z,,, ", -1]
Fullerton, CA 92831	92831
Kamloops, BC 3J2 X7G	3J2 X7G

The astute reader may have realized that a simpler text funnel can do the same job, `["0-9", -1]`. Of course it would not have illustrated the `≠` feature. The moral of the story is: watch out for college solutions when a grade school solution may work just as well!

A text funnel that uses the `≠` symbol can also work in reverse gear, so that it specifies the last character that does not match, instead of the first. The `≠` symbol must be first, and then the `-` symbol. For example, here is yet another formula for extracting the price from [Line](#).

```
Line["≠-0-9,.-",-1][2,-1]
```

Here's how this formula would process some of our favorite sample text strings.

Original Data	After <code>["≠-0-9,.-",-1][2,-1]</code>
Tape Cartridge \$22.95	22.95
Data Cassette 7.80	7.80
XB3 Cleaner \$19.50	19.50

Unlike some of our previous examples, this formula does not rely on a `$` symbol or a space in front of the price, and it does not choke if there is a number in the item description.

Limitations of Text Funnel

Unlike Humpty Dumpty, text items are easy to put together but hard to take apart intelligently. Text funnels are a powerful tool, but they do have limitations. One limitation is that, by themselves, they can only work with one character at a time. If you want to start stripping text with the word `fax` or `P.O. Box` a text funnel can't do it on its own. You'll have to combine the funnel with the `search()` function for jobs like this (see [reference page 5707](#)).

The most important limitation of text funnels is that they cannot work reliably if there is not a single consistent pattern in the data. If the data has no pattern at all, you're out of luck (short of re-keying data). If the data has two or more patterns you'll need to isolate each pattern and process each one with a separate text funnel. One way to do this is with the `?(` function (see [reference page 5008](#)). This formula extracts the local phone number from a complete phone number. If the complete phone number starts with `(`, the formula uses a text funnel that strips out the area code, otherwise the local number starts with the first character.

```
?( Phone[1,1]="(" , Phone[7;8] , Phone[1;8] )
```

Here's how this formula would process some of our favorite sample text strings.

Original Data	After <code>?(Phone[1,1]="(",Phone[7;8],Phone[1;8])</code>
(714) 555-1212	555-1212
852-9632	852-9632
(562) 492-1438 ext 23	492=1438

Don't be afraid to combine text funnels with other functions and statements. Some functions that are often useful with text funnels include `?(` (see [reference page 5008](#)), `length()` (see [reference page 5467](#)), `strip()` (see [reference page 5802](#)), `stripchar()` (see [reference page 5802](#)), `search()` (see [reference page 5707](#)), `replace()` (see [reference page 5665](#)), and `array()` (see [reference page 5036](#)).

String Testing Functions

These functions return information about the content of a string.

Function	Reference Page	Description
cardvalidate(text)		This function returns true if the text contains a credit card number that is valid, false if it is not. Credit cards have an internal checksum that allows a number to be validated for simple data entry errors (for example missing or transposed digits). This function checks to make sure that a number is a valid credit card number. Of course the function cannot tell whether this card number has actually been issued, or what the credit limit is or any other financial information about the card. It simply provides a simple check for missing or transposed digits. If this function says the card number is not valid you are sure that the number is wrong, but if the function returns true you would still need to check with the issuer to determine if this is a valid card. (Note: Unlike the cardvalidate statement, this function will remove any non-numeric data from the card number, so it's ok to leave in spaces, dashes, etc.)
checkenglish(word)		This function returns true if the specified word is in Panorama's English dictionary, false otherwise. (Note: If the optional Panorama Spell + Zip package is not installed this function will always return false.)
length(string)	Page 5467	This function counts the number of characters in a string. The result is an integer. If the string is empty, the result will be zero.
linecount(string)		This function counts the number of lines in the text.
rangecontains(thetext,therange)		This function checks to see if the text contains any characters in the specified range. The range must be a series of character pairs, for example AZ for upper case alphabetic characters, AZaz for upper and lower case, 09 for numeric digits, etc. If the text contains any characters in the specified range the function returns true, otherwise it returns false. For example, rangecontains(Company,"09") will return true if the company name contains any numeric digits, false if it doesn't.
rangematch(string,range)		This function checks text to see if the text matches the specified range. The range must be a series of character pairs, for example AZ for upper case alphabetic characters, AZaz for upper and lower case, 09 for numeric digits, etc. If it matches the function returns true, if it doesn't match, it returns false. For example, rangematch(Address,"AZaz09 ") will return true if the address contains only letters, numbers and spaces, false if it contains any other characters.
search(string,phrase)	Page 5707	This function searches through a string looking for a word or phrase. If the search is successful, the function returns the position of the phrase within the string, otherwise the function returns zero. For example, the formula search(Name,"Dr.") will return a non-zero value (usually 1) if the name contains Dr. , or zero if it does not.

Function	Reference Page	Description
sizeof(name)	Page 5777	<p>This function calculates the amount of memory used by a field cell or a variable. Name is the name of the field or variable that you want to calculate the size of. The function returns the number of bytes of memory used by the variable or field cell.</p> <p>The sizeof(function can be used to decide if a numeric or date field is empty or not. The example procedure shown below selects all the records with no price (not the same as records with a price of zero).</p> <pre>select sizeof(Price)=0</pre> <p>Another use for the sizeof(function is to check if a variable is taking up too much memory. This example checks to see if the variable importLetter is more than 500 bytes long. If it is, the procedure clears the variable.</p> <pre>if sizeof(importLetter)>500 importLetter=" " endif</pre>
wordcount(string)		This function counts the number of words in the text.

String Modification Functions

These functions modify the contents of a string. Usually the string is actually a database field. Remember, to use a database field as a string parameter simply use the name of the field, for example `upper(Name)`. You'll often want to use these functions to modify the existing data in a field. For example, you might want to convert all company names to upper case. To convert existing data use the **Manipulate Data in Field** command in the Fields Menu (see "[Starting with a Formula](#)" on page 439 of the *Panorama Handbook*). This command calculates the formula over and over again—once for each selected record.. Note: In addition to the functions listed here you will also find methods for modifying strings in "[Functions for Taking Strings Apart](#)" on page 68, "[Taking Strings Apart \(Text Funnels\)](#)" on page 69, "[Text Arrays](#)" on page 93 and "[HTML Tag and Tag Parsing Functions](#)" on page 101.

Function	Reference Page	Description
<code>applescriptstring(string)</code>		This function converts the string into an AppleScript string literal. It surrounds the text with double quote characters, and escapes any double quote and/or backslash characters within the text. This function is designed to be used with the <code>executeapplescript</code> statement.
<code>batchreplace(text,array,sep,subsep)</code>		This function performs multiple find and replace operations on a string of text. The array must be two dimensional, it specifies what words or phrases are going to be replaced and with what. This is similar to <code>replacemultiple()</code> but the before and after strings are combined into a single array, making it easier to use.
<code>connect(prefix,connector,suffix)</code>		This function appends a prefix and suffix together with a connector in between. If either the prefix or the suffix is missing then the connector will also be left out. For example, <code>connect(City," ",State)</code> combines the city and state with a comma and space in between, but if either the city or state is missing then the comma and space will also be left out. See also the <code>sandwich()</code> and <code>yoke()</code> functions in this table.
<code>crtovtab(string)</code>		This function converts carriage returns (ASCII 0x0D) into vertical tabs (ASCII 0x0B). Some programs (including Panorama) will convert vertical tabs into carriage returns when importing, allowing individual data cells to contain carriage returns.
<code>defaulttext(text,default)</code>		This function returns the text value supplied in the first parameter. However, if this text value is empty ("") the function will return the specified default value.
<code>extract(text,separator,item)</code>	Page 5211	<p>This function extracts a single data item from a text array. This function is almost identical to the <code>array()</code> function. The <code>extract()</code> function is excellent for extracting a word, line or phrase from a larger text item. It can also be used to count the number of items in the array. There are three parameters: text, separator and item. Text is the item of text that contains the data you want to extract. Separator is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see "Special Characters" on page 57). For tab delimited arrays use the - character. Item is the number of the data item you want to extract. The first item is item 1, the second is item 2, etc.</p> <p>Using an item number of -1 tells the <code>extract()</code> function to count the number of data items in the array. This is similar to the <code>arraysize()</code> function. In this case the <code>extract()</code> function will return a number, not text.</p> <p>If the item parameter is 1 or greater, this function returns an item of text from the array. Only the item itself is returned, the separator characters on each end are not included. If the item does not exist (for example if you ask for item 12 from a 7 item array) the function will return empty text ("").</p>
<code>fixedwidth(string,width)</code>		This function makes the text a fixed width. If the text is shorter than the specified width, it is padded with spaces. If it is longer than the specified width, it is cut off.

Function	Reference Page	Description
<code>fixedwidthright(string,width)</code>		This function makes the text a fixed width. If the text is shorter than the specified width, it is padded with spaces on the left (i.e. the text is right justified). If it is longer than the specified width, it is cut off on the left.
<code>linestrip(text)</code>		This function removes any blank lines from the text.
<code>lower(string)</code>	Page 5516	This function converts all of the letters in the string to lower case. For example, the formula <code>lower(Terms)</code> will convert <code>NET 30</code> to <code>net 30</code> , or <code>C.O.D.</code> to <code>c.o.d.</code> See also the upper and upperword functions.
<code>obscuredigits(number,count)</code>		This function obscures digits (usually a credit card number) with X's. The first parameter is the text that contains the digits. The second parameter is the number of digits on the end that will NOT be obscured. For example the formula <code>obscuredigits("1234-5678-9876-5432",4)</code> will produce the value <code>XXXX-XXXX-XXXX-5432</code> . Notice that the function retains any additional formatting in the text, in this case dashes.
<code>onespace(string)</code>		This function removes any extra spaces between words, so that there is exactly one and only one space between each word.
<code>onewhitespace(string)</code>		This function removes any extra whitespace between words, making sure that there is one and only one space between each word. Other whitespace characters (carriage returns, tabs) are converted to spaces and removed if there is more than one between words.
<code>padzero(text,width)</code>		This function makes the text a fixed width. If the text is shorter than the specified width, it is padded with 0's on the left (i.e. the text is right justified). If it is longer than the specified width, it is cut off on the left..
<code>quoted(string)</code>		This function surrounds the supplied text with double quote characters. If the text contains any double quotes they will be doubled, making this a legal string constant. This function is typically used in conjunction with the <code>execute</code> and <code>executeapplescript</code> statement.
<code>randomletter(option)</code>		This function returns a random letter. If the option is "U" then the letter will be between A and Z. If the option is "L" then the letter will be between a and z. If the option is any other value then the result may be either A-Z or a-z.
<code>randomline(text)</code>		This function picks a random line from some text.
<code>randomword(wordlist)</code>		This function picks a random word from some text.
<code>rep(string,count)</code>	Page 5662	This function replicates a string over and over. The number of replications is specified by the count (a number). This function is handy for creating a long repeating string. For example to create a string containing twenty asterisks in a row, use the formula <code>rep("*,20)</code> . The count does not have to be a constant, but it must be an integer.
<code>replace(string,search,replace)</code>	Page 5665	This function searches for a word or phrase within a string and if found, replaces it with a new word or phrase. The first parameter is the string that may contain the word or phrase. Usually this parameter is a database field. The second parameter is the word or phrase to search for. The third parameter is the new word or phrase. For example, to replace <code>Corporation</code> with <code>Corp.</code> in the <code>Client</code> field, use the formula <code>replace(Client,"Corporation","Corp.")</code> . To use this formula to replace the data in the database, use the Formula Fill command. (For a simple replace case like this, however, it is easier to use the Change command. The <code>replace()</code> function is useful when you want to perform other transformations in addition to the <code>replace()</code> .)

Function	Reference Page	Description
<p style="text-align: center;">replacemultiple(string,search,replace,sep)</p>	<p style="text-align: center;">Page 5667</p>	<p>This function searches for a set of words or phrases and replaces them with another set of words or phrases. It is similar to the <code>replace()</code> function, but can replace a whole set of items at once. The string parameter must contain the text that contains the words or phrases you want to replace. The search parameter contains a list of words or phrases to search for. The items in this list must be separated by the sep character. Here's an example that uses comma as the separator.</p> <pre style="text-align: center;">"Drive,Lane,Avenue,Boulevard"</pre> <p>The replace parameter contains another list of words or phrases. These must use the same separator character and be in the same order as the search parameter. Here's another example.</p> <pre style="text-align: center;">"Dr , Ln , Ave , Blvd"</pre> <p>Putting it all together, here's an example that inserts abbreviations in an address.</p> <pre style="text-align: center;">replacemultiple(Address, "Drive,Lane,Avenue,Boulevard", "Dr , Ln , Ave , Blvd", ",")</pre> <p>In this example we've separated each parameter onto a separate line, but this is not necessary. Also, keep in mind that you can use any character as a separator, not just a comma.</p>
<p style="text-align: center;">sandwich(prefix,root,suffix)</p>	<p style="text-align: center;">Page 5689</p>	<p>The <code>sandwich()</code> function assembles a text item from three smaller text items. The prefix and suffix are slapped on the ends of the root, just like a sandwich. However, if the root is empty, the prefix and suffix are also left off (the result is an empty text item), just as you wouldn't make a sandwich without any meat.</p> <p>Suppose you have a database with names and titles, and you want to display this information in a report with the titles surrounded by parentheses. The formula below could be used with an auto-wrap text object or Text Display SuperObject.</p> <pre style="text-align: center;">Name+sandwich(" (",Title," ")</pre> <p>If the person has a title it will appear in parentheses like this: Steve Johnson (Sales Mgr). If they don't have a title then no parentheses will appear. The <code>sandwich()</code> function is useful any time you have optional data items combined together with punctuation in between. See also the <code>connect()</code> and <code>yoke()</code> functions in this table.</p>
<p style="text-align: center;">strip(text)</p>	<p style="text-align: center;">Page 5801</p>	<p>This function strips off leading and trailing blanks and other whitespace (carriage returns, tabs, etc.) This function has one parameter, the item of text that you want to strip. The function removes blanks at the beginning or end of the text, but does not affect blanks in the middle of the text. It also removes carriage returns, tabs, or any character with an ASCII value less than 32.</p>

Function	Reference Page	Description
stripchar(text,range)	Page 5802	<p>This function removes characters you don't want from a text item. You specify exactly what kinds of characters you want and don't want included in the final output. Text is the item of text that you want to strip. Range specifies what kinds of characters you want to keep and what kinds of characters you want to strip away. The range consists of one or more pairs of characters. Each pair specifies a set of characters you want to keep. For example, the pair AZ means that you want to keep the characters from A to Z. For alphanumeric characters the set is pretty obvious. For other types of characters you should check an ASCII chart (see "Characters and ASCII Values" on page 87). For example the pair #& specifies a set of four characters: #, \$, % and &. You can use the ASCII Chart wizard to try out your character ranges, see "Showing Character Ranges with the ASCII Wizard" on page 91.</p> <p>If a pair consists of the same character repeated twice in a row, the set is just that single character. For instance the pair ## means you want to keep one character: #.</p> <p>The range may consist of several pairs put together. For example the range AZaz09.. consists of four pairs, and specifies that all letters, numbers, and periods will be kept, with all other characters stripped away.</p> <p>One handy use for this function is to quickly check if a field or variable contains any inappropriate characters. If a field or variable changes when you run it through the stripchar(function it must contain characters that are not part of the specified range.</p>
striptmltags(text)		This function removes all HTML tags from the text.
stripprintable(text)		This function removes any non-displayable characters from the text.
striptoalpha(text)	Page 5805	<p>This function removes everything but alphabetic letters from a text item. Everything else (numbers, spaces, punctuation, non-English letters, etc.) will be removed from the text.</p> <p>One handy use for this function is to quickly check if a field or variable contains all alphabetic characters. If a field or variable changes when you run it through the striptoalpha(function it must contain non-alphabetic characters.</p>
striptonum(text)	Page 5807	<p>This function removes everything but numeric digits from a text item. Everything else (letters, spaces, punctuation, non-English letters, etc.) will be removed from the text.</p> <p>One handy use for this function is to quickly check if a field or variable contains all numeric digits. If a field or variable changes when you run it through the striptonum(function it must contain non-numeric characters.</p>
upper(string)	Page 5880	This function converts all of the letters in the string to upper case. For example, the formula upper(Terms) will convert net 30 to NET 30 , or c.o.d. to C.O.D. See also the lower(and upperword(functions.
upperword(string)	Page 5881	The upperword(function converts the first letter of each word in the string to upper case, and all other letters to lower case. For example the formula upperword(State) will convert new york to New York , or will convert VERMONT to Vermont . See also the lower and upper functions.
vtabtocr(string)		This function converts vertical tabs (ASCII 0x0B) into carriage returns (ASCII 0x0D). Some programs (including Panorama) will convert vertical tabs into carriage returns when importing, allowing individual data cells to contain carriage returns. After the import you can use this function to turn the vertical tabs back into carriage returns.
yoke(prefix,joiner,suffix)		This function appends two text items (prefix and suffix) together. If both are non-blank, a joiner is placed in between. If either (or both) is blank, the joiner is not used. In some ways this is the reverse of the sandwich(function.

Converting Between Numbers and Strings

These functions convert numbers into strings and strings into numbers.

Function	Reference Page	Description
asc(string)	Page 5060	This function converts the first character of the string into a number based on the ASCII value of the character. For example the formula <code>asc("Y")</code> returns the value 89, while <code>asc("Z")</code> returns the value 90. See also the <code>chr()</code> function.
bytepattern(number)		This function converts a number into text. The number is treated as a number of bytes, and depending on the size will be displayed as bytes, kilobytes, megabytes, or gigabytes. This function uses SI units, meaning that 1 kB = 1000 bytes, 1 MB = 1000 kB, and 1 GB = 1000 MB. See http://en.wikipedia.org/wiki/Megabyte for more information on SI units for data size.
chr(number)	Page 5099	This function converts a number into a single character of text based on the ASCII value of the number. The number should be an integer between 0 and 255. For example, the letter A has an ASCII value of 65, while the letter B is 66. You can create special characters with this function; TAB is 9 and RETURN is 13. See also the <code>asc()</code> function.
commastr(number)		This function converts a number into text, with a comma every third digit. The number is converted as an integer, with no places after the decimal point.
dollarsandcents(number)		This function converts a number to text formatted as dollars and cents (for example 98123.45 becomes Ninety eight thousand one hundred twenty three dollars and 45 cents).
exportcell(value)	Page 5209	This function converts a value into text without any special formatting. For numeric values this function is the same as the <code>str()</code> function (see below). The advantage of this function is that it works with any kind of value - text, numeric or date. Use this function when for some reason you don't know what kind of data you need to convert. (Important Note: If the field is a text field, only the first line of the original text will be returned by this function. Any additional lines will be stripped off.)
hex(string)		Converts text with hex characters into a number. For example the value of <code>hex("0C2")</code> is 194.
hexbyte(number)		Converts a number to text formatted as a two digit hexadecimal number. For example the result of <code>hexbyte(68)</code> is 44.
hexlong(number)		Converts a number to text formatted as a eight digit hexadecimal number. For example the result of <code>hexlong(68)</code> is 00000044.
hexstr(number)		Converts a number to text formatted as a hexadecimal number. For example the result of <code>hexstr(68)</code> is 00000044. This function can also accept binary values with more than 4 bytes (see " Raw Binary Data " on page 156).
hexword(number)		Converts a number to text formatted as a four digit hexadecimal number. For example the result of <code>hexword(68)</code> is 0044.
money(number)		Converts a number to text, formatted with commas every three digits and two digits after the decimal point (for example 98,123.45).
nth(number)		This function converts a number into an ordinal, i.e. 1=1st, 2=2nd, 3=3rd, 4=4th, etc.
pattern(number,string)	Page 5599	This function converts a number into text, using the string as an output pattern. For example the formula <code>pattern(Price,"\$#.,##")</code> will convert the price 3458.23 into the string \$3,458.23. The pattern adds the \$ and the comma. For more information on numeric output patterns see " Numeric Output Patterns " on page 250 of the <i>Panorama Handbook</i> .

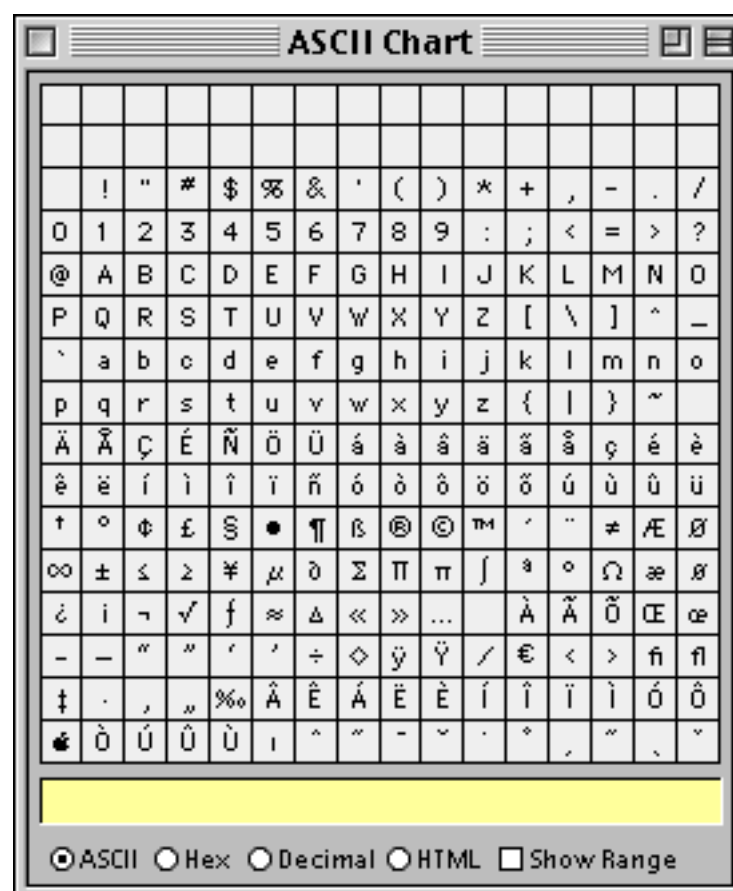
Function	Reference Page	Description
places(number,places)		This function converts a number to text with a specific number of places after the decimal point. The value is truncated (not rounded) to the number of places specified..
radix(radix,text)	Page 5626	<p>This function converts a text item containing a hex, octal, or binary number into a standard Panorama number (decimal). See "NON DECIMAL NUMBERS" on page 5543 of the <i>Panorama Reference</i> for background information on hex, octal and binary numbers. Radix is the base for the numbering system you are converting from. Legal radix values are 2, 4, 8, 16 or 32. Or you can specify the radix as "binary" (same as 2), "octal" (same as 8) or "hex" (short for hexadecimal, same as 16). Text is a text item that contains the non-decimal number you want to convert. This function normally returns an integer that contains the decimal (base 10) number corresponding to the hex, octal, or binary number input to the function.</p> <p>If the radix is hex and there are more than 8 digits in the input text, or if the radix is binary and there are more than 32 digits, this function will return a raw binary value instead of a number. This binary value may be of unlimited length. Like all binary values, it cannot be calculated with, but should be handled as a text item.</p>
radixstr(radix,number)	Page 5628	<p>This function converts a number into a text item containing the equivalent hex, octal, or binary number. See "NON DECIMAL NUMBERS" on page 5543 of the <i>Panorama Reference</i> for background information on hex, octal and binary numbers. Radix is the base for the numbering system you are converting from. Legal radix values are 2, 4, 8, 16 or 32. Or you can specify the radix as "binary" (same as 2), "octal" (same as 8) or "hex" (short for hexadecimal, same as 16). Number is the number you want to convert to hex, octal, or binary. If the radix is 2, 16, "binary", or "hex" the number can be a raw binary data (text) value. This function returns a text item that contains the hex, octal, or binary number equivalent to the number (or binary data) passed to the function. The first example converts the decimal value 256 to hexadecimal.</p> <p><code>radixstr(16,256)</code></p> <p>This function will calculate that 256_{10} is 100 hex.</p> <p>Here is another example:</p> <p><code>radixstr("binary",5)</code></p> <p>This will calculate that 5_{10} is 0000000000000000000000000000000101 binary.</p>
scientificnotation(number)		Converts a number to text, formatted in scientific notation with three places after the decimal point (for example 9.812e+4).
str(number)	Page 5799	This function converts a number into text without any special formatting. If you want to format the number (add commas, set # of digits, etc.) use the pattern(function.
val(string)	Page 5886	This function converts a string into a number. The string must start with one or more numeric digits. Everything after the first non-numeric character will be ignored. For example, the formula <code>val(Address)</code> will return the number 731 if the address is 731 N. Miller St.

Function	Reference Page	Description
zbpattern(number,pattern)		This function displays a number using a pattern. Unlike the normal pattern() function, the zbpattern() function will output "" if the number is zero. (Note: zb is short for zeroblank.)
exportcell(field)	Page 5209	<p>This function takes any database field and converts it to text, using the appropriate pattern if one has been defined in the design sheet. Field is the name of the field to be converted to text.</p> <p>The function always returns a text type data item. The power of the exportcell() function is that it does not require you to know what type of data you are exporting. It simply takes whatever kind of data is in the field (text, number, date, whatever) and converts it into text.</p>

Characters and ASCII Values

Just as molecules are built from atoms, text is built from characters. And like an atom which can be divided into electrons, protons, and neutrons (among others), characters also have an internal structure. Just as with atoms, the internal structure of characters can usually be ignored, and you may want to skip the following section if you are a beginner. Sometimes however, knowledge of the internal structure of characters can be very helpful.

On most computer systems there are 256 possible characters. (Some Japanese and Chinese systems allow thousands of characters, however Panorama does not currently support this.) Each character has a number from 0 to 255. Of these 256 characters, about 200 are associated with symbols (letters, digits, punctuation, etc.). For example, the symbol for the letter A is represented by character number 65. You can use the **ASCII Chart** wizard (in the Developer Tools subfolder of the Wizard menu) to see a complete list of all 256 characters and their symbols.



The numbers have not been assigned to symbols arbitrarily, but have been assigned using a system called ASCII. The number associated with a character is called the ASCII value of the character. (For you technoweenies, ASCII stands for American Standard Computer Interchange Interface.) If you look at the ASCII table on the next page you'll notice that the characters with ASCII values from 0-31 have no symbols. These characters are used for special keys like return, tab, and enter. ASCII value 32 is the space character, then we have some punctuation. ASCII values 48 through 57 are the numeric digits 0 through 9, in order. ASCII values 65-90 are the upper case letters A through Z, in alphabetical order. ASCII values 97-122 are the lower case letters a through z, again in alphabetical order.

Panorama uses the ASCII values of characters when it compares two text items to see which is larger or smaller. Since the ASCII value of B (66) is greater than the ASCII value of A (65), the text item B is “larger” than A. However, the ASCII value of a (97) is greater than B (66), so the text item a is “larger” than B. You have to watch out for this problem whenever you compare text that is a mixture of upper and lower case.

Working with Character Values

Usually it's not necessary to worry about the numeric value of a particular character—you can just think of it as a character. However, if you want to perform any kind of math on the character itself it is necessary to convert the character in to a number. For example you can add one to a character value to get the next character value (A → B → C etc.). Or you can calculate the number of characters between two characters.

Panorama has two special functions that allow you to work with character values directly. The `asc()` function converts a character to its ASCII value. The `chr()` function converts an ASCII value to the corresponding character.

The following example procedure asks the user to enter a range of characters, for example A-F. It uses the `asc()` function to convert the characters into the corresponding ASCII numeric values, then calculates the number of characters in the range.

```
local LetterRange, StartLetter, EndLetter, LetterCount
LetterRange=""
gettext "Enter character range:", LetterRange
StartLetter=LetterRange[1,1]
EndLetter=LetterRange[-1,-1]
LetterCount=abs(asc(EndLetter)-asc(StartLetter))
message LetterRange+": "+pattern(LetterCount+1, "# character~")
```

If the person enters **A-F** the procedure will display **A-F: 6 characters**.

The next example procedure is similar but actually displays a list of the characters in the range. It uses the `chr()` function to convert the numbers back into characters.

```
local LetterRange, StartLetter, EndLetter
local LetterCount, LetterBump, Letters
LetterRange=""
gettext "Enter character range:", LetterRange
StartLetter=asc(LetterRange[1,1])
EndLetter=asc(LetterRange[-1,-1])
LetterCount=EndLetter-StartLetter
LetterBump=LetterCount/abs(LetterCount)
Letters=""
loop
  Letters=Letters+chr(StartLetter)
  StartLetter=StartLetter+LetterBump
while StartLetter≠EndLetter
message LetterRange+": "+Letters
```

If the person enters **A-F** the procedure will display **A-F: ABCDEF**. If the person enters **F-A** the procedure will display **F-A: FEDCBA**.

Warning: Don't confuse the `asc()` and `chr()` functions with the `val()` and `str()` functions. The `asc()` and `chr()` functions convert single characters based on their ASCII values. The `val()` and `str()` functions convert entire text items based on the number the characters spell out. For example `asc("4")` is 52, because 52 is the ASCII value of the character "4." On the other hand, `val("4")` is 4. Confused? You almost certainly want to use `val()` and `str()` unless you are sure you know what you are doing.

Invisible Characters

The ASCII system contains a number of characters that are normally invisible. In fact, every ASCII character with a value of 32 or lower is invisible. Normally you will not be concerned with invisible characters. However, there are three special invisible characters that do get a lot of use: **space**, **carriage return**, and **tab**.

The space character (ASCII value 32) is not quite invisible, because it does take up space. You can easily enter this value by pressing the **Space Bar**. In a formula you can enter a space directly [" "] or using the `chr()` function [`chr(32)`].

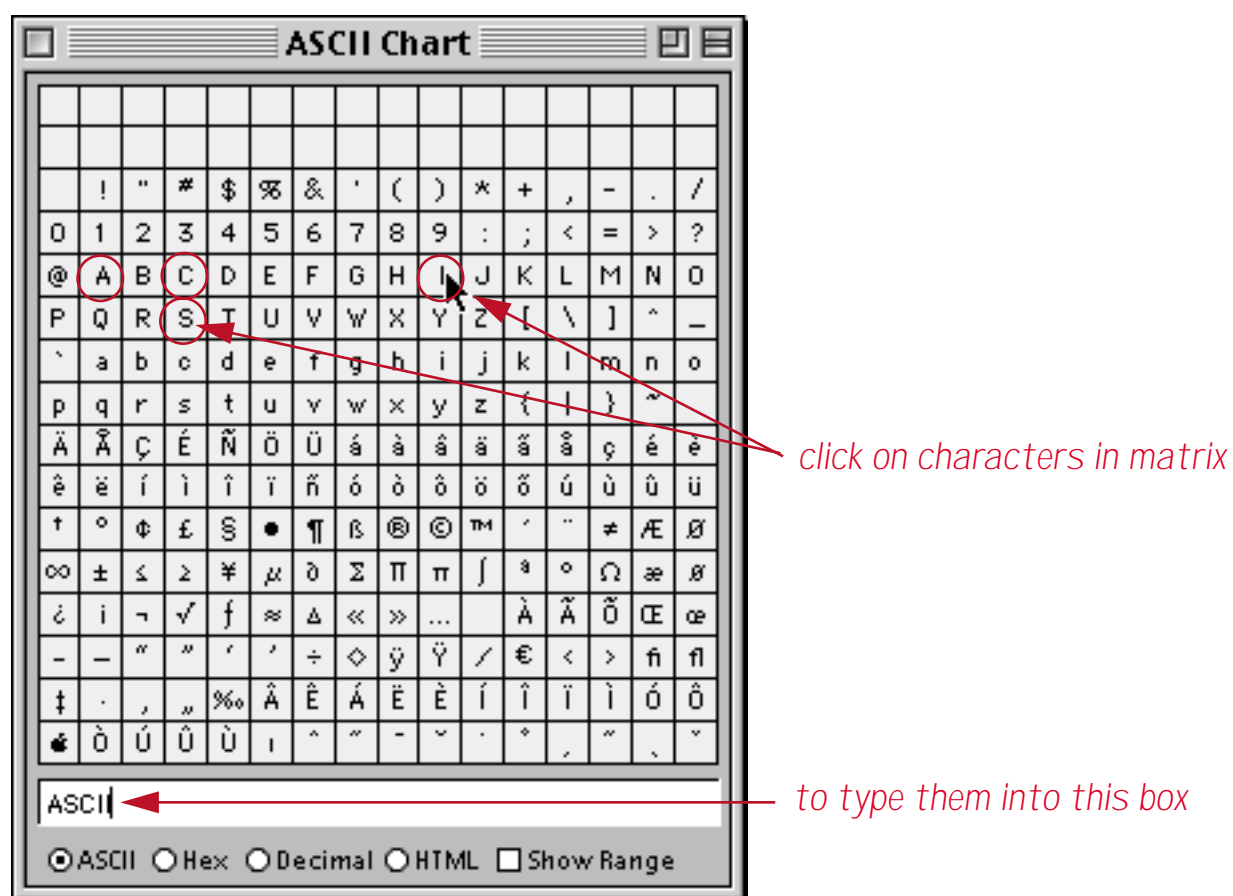
The carriage return character is used to start a new line of text. This character has an ASCII value of 13. You can enter this value into a formula using the ¶ symbol (see "[Special Characters](#)" on page 57) or as `chr(13)`.

(Trivia question: why is this character called carriage return? In a few years probably no one will remember. In case you are already too young to remember, typewriters (and teletypes) used to place the paper on a carriage that moved back and forth as you typed. When you pressed the **Return** key the carriage would “return” back to the beginning of the line and also advance down to the next line, hence carriage return. In fact, on old manual typewriters this was accomplished with a lever, not a key.)

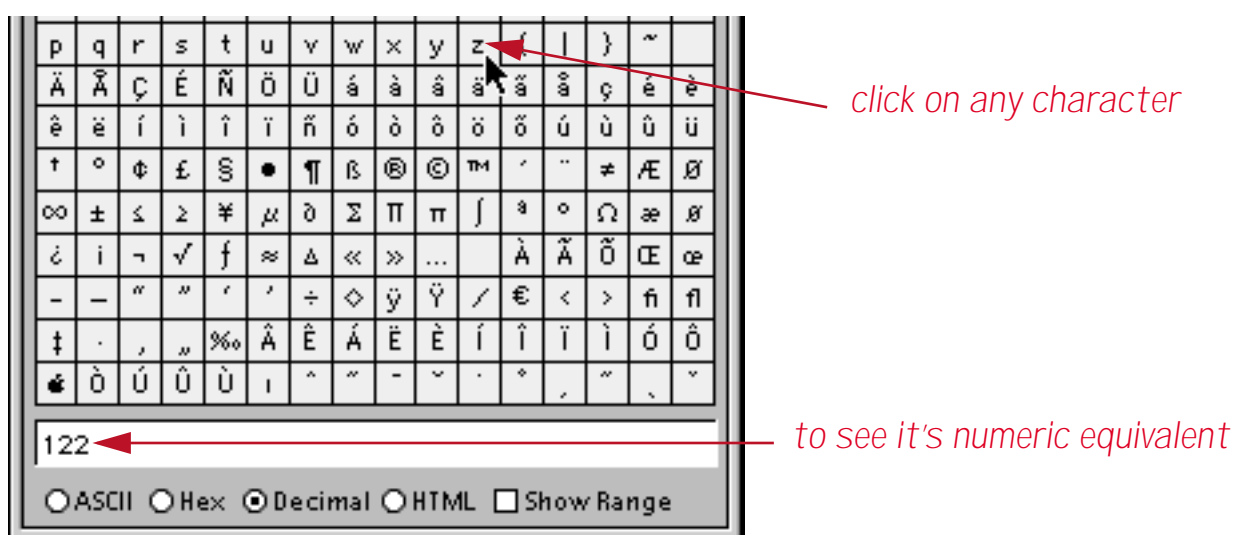
The **tab** character is usually not found inside data, but is often found in text files created by editors or word processors (including the Panorama word processor). The tab character has an ASCII value of **9**. You can enter this value into a formula using the **↵** symbol (see “[Special Characters](#)” on page 57) or as **chr(9)**.

The ASCII Chart Wizard

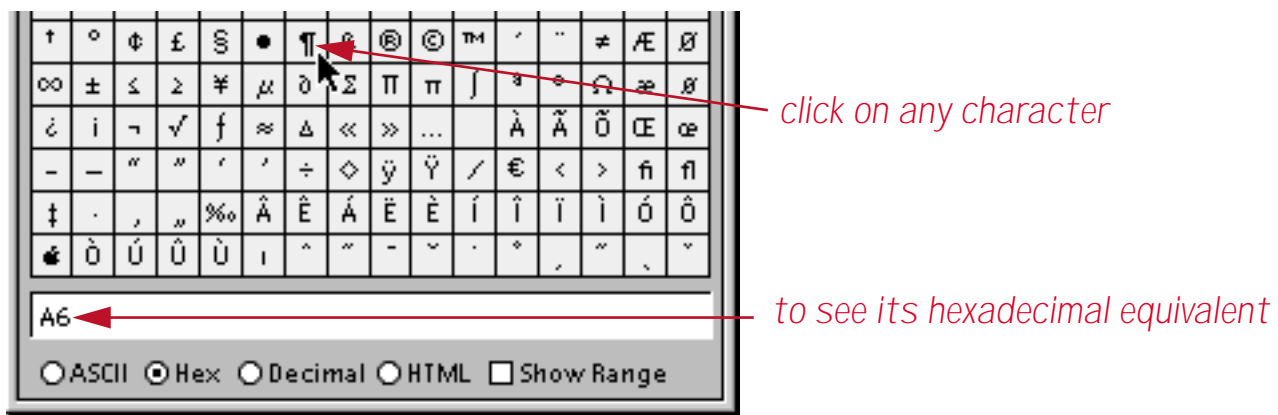
The **ASCII Chart** wizard allows you to display a matrix showing all 256 ASCII characters. When you click on a character it types that character into the box at the bottom.



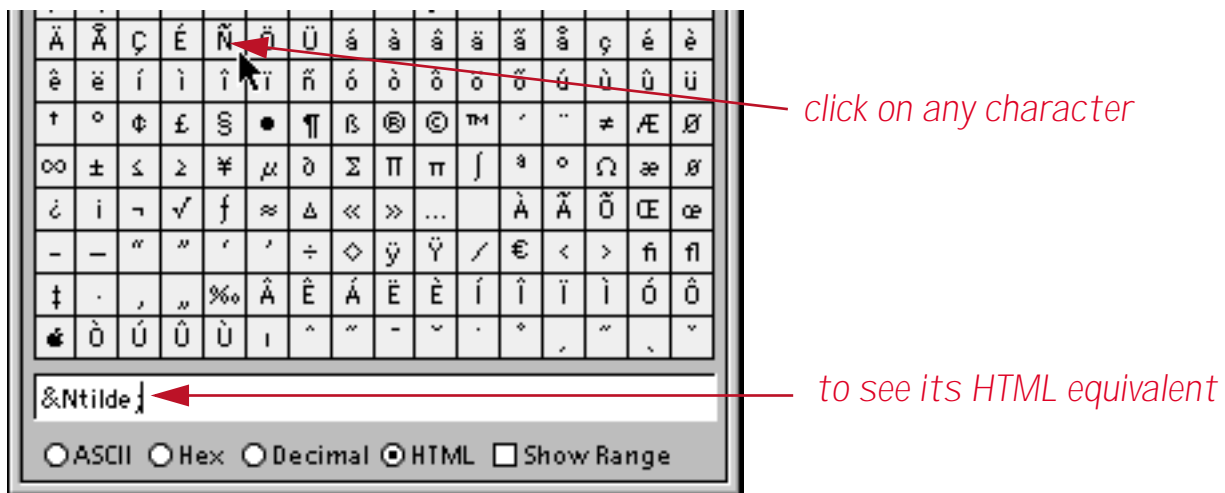
If you select the **Decimal** option then clicking on a character enters the corresponding numeric value of the character into the box.



Use the **Hex** option to see the numerical value of the character in hexadecimal (see “[Raw Binary Data](#)” on page 156).

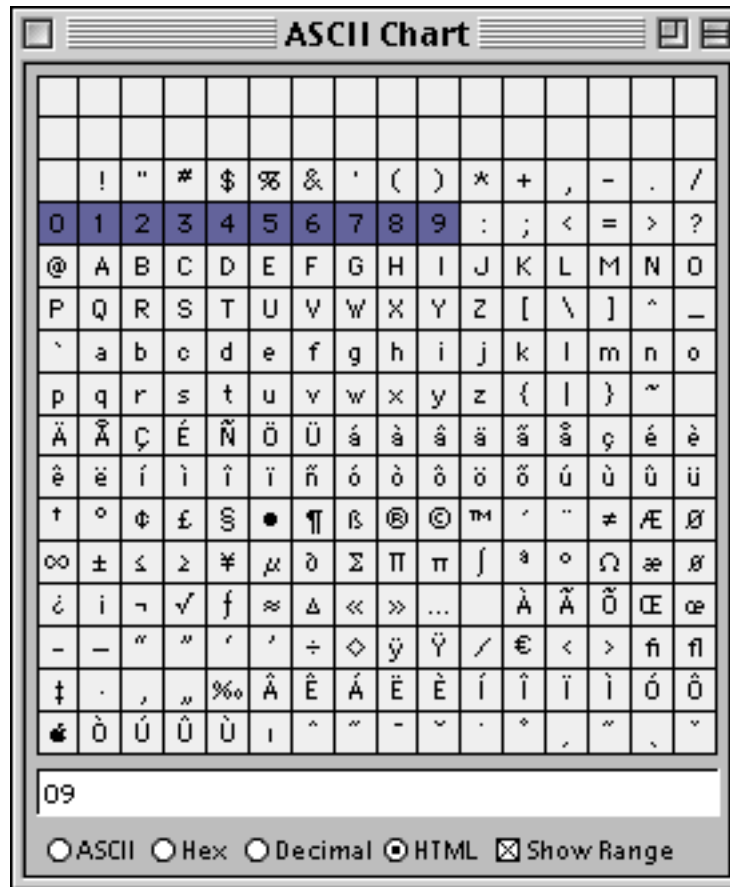


HTML has special codes for many characters. Use the **HTML** option to see the equivalent code (if any) for a special character.

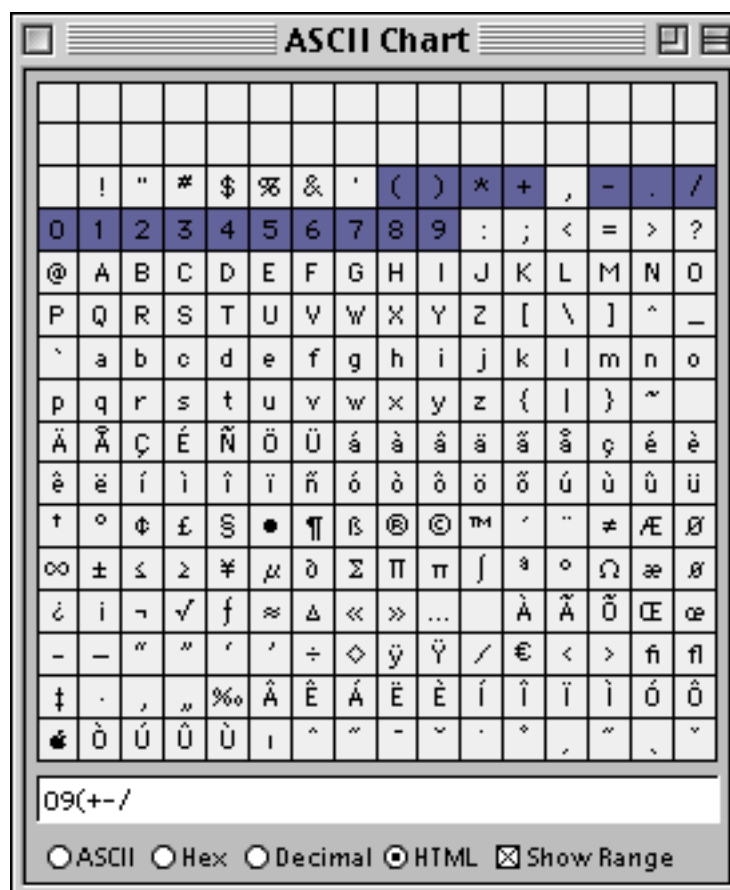


Showing Character Ranges with the ASCII Wizard

Several Panorama features use character ranges, including field properties (see “[Restricting Character Types](#)” on page 293), text funnels (see “[Taking Strings Apart \(Text Funnels\)](#)” on page 69) and the `stripchar()` function (see “[String Modification Functions](#)” on page 80). The **ASCII Chart** wizard allows you to preview a character range by selecting the **Show Range** option and typing the range into the box. All of the characters in the range will be highlighted in blue. For example, the illustration below shows the range **09**, which includes all numeric characters.



Here is a more complex range that includes all the characters used in basic mathematical formulas.



You can use the **ASCII Chart** wizard to try out your ranges before you actually use them in a database.

ASCII Character Constant Functions

These functions return common ASCII characters.

Function	Description
<code>info("lineseparator")</code>	This function returns the line separator character on the current platform. On Macintosh systems this is a carriage return. On Windows PC systems this is a carriage return followed by a linefeed (CR-LF).
<code>cr()</code>	This function generates a carriage return. This is equivalent to <code>chr(13)</code> and is also the same as ¶.
<code>crlf()</code>	This function generates a carriage return line feed. This is equivalent to <code>chr(13)+chr(10)</code> .
<code>lf()</code>	This function generates a line feed. This is equivalent to <code>chr(10)</code> .
<code>tab()</code>	This function generates a tab character. This is equivalent to <code>chr(9)</code> and is also the same as ↹.
<code>vtab()</code>	This function generates a vertical tab character. This is equivalent to <code>chr(11)</code> .

Text Arrays

An array is a numbered collection of data items. Panorama includes a number of functions and statements that treat a single text data item as if it were a numbered collection of smaller items. The smaller text data items must be separated from each other by a delimiter, for instance a comma or carriage return.

Consider the text data item shown below. Panorama would normally treat this as a single item with a length of 40 characters. The functions described in this section, however, can treat this text as a collection of 7 elements separated by semicolons.

```
white;red;orange;yellow;green;blue;black
```

In this example, the ; is the separator character. You can use any character you want for a separator character, in fact, you can use different separator characters at different times. You could even build a multi-level array by using two different separator characters.

Using the array functions and statements provided by Panorama you can extract elements from an array, change array elements, even sort an array. Since arrays are really text, they can be stored in any variable or any text field, and they can be edited with the data sheet, a data cell, or a Text Editor SuperObject.

There are many statements and user interface elements that work with text arrays, including lists and pop-up menus. There are also a number of functions that generate text arrays, including functions for building lists of files, windows, fields, choices, and data. Most of these statements, user interface elements, and functions require that carriage returns be used as separators, so that each array element is on a separate line.

It is up to you to keep track of the fact that you are using an array and what the separator character is. Panorama won't stop you from trying to access the array of colors above as if it were delimited with commas instead of semicolons, but you probably won't get the results you wanted unless you use the correct separator character.

(If you are familiar with the arrays in C or Pascal, Panorama text arrays are quite a bit different, although both are a numbered collection of items. As with anything unfamiliar, Panorama text arrays probably won't look as good as the ones you are used to at first. Panorama arrays do have some significant advantages though: they don't have to be declared in advance, each array element can be of unlimited length without wasting space, and Panorama arrays can be directly edited. It's also very easy to "pre-fill" a Panorama array with a list of values.)

Picking a Separator Character

Any ASCII character can be used as a separator character, so you have 256 possible choices. Common separators include comas, semicolons, slashes, carriage returns, spaces and tabs.

It's important to pick a separator character that will not occur in the data elements of your array. If your data may include commas, don't use the comma as a separator character. If the data might include carriage returns, don't use a carriage return. If you want to be extra sure to avoid conflicts, pick a non-printing character. You can use the `chr()` function ([reference page 5099](#)) to generate non-printing characters, for example `chr(1)`, `chr(2)`, `chr(3)`. Most `chr()` values below 32 are non-printing except for `chr(9)` and `chr(13)`, which correspond to tab and carriage return.

Some Panorama user interface elements and functions use text arrays as parameters or to hold a list of values. For these applications the separator character is usually required to be a carriage return. For example, the Pop-Up Menu SuperObject uses a carriage return delimited array to define the list of pop-up menu choices (see "[The Pop-Up Menu Formula](#)" on page 863 of the *Panorama Handbook*). The `lookupall()` function ([reference page 5502](#)) extracts information from another database and places it into an array with whatever separator you specify. Consult the documentation for each individual statement, function or SuperObject to see the exact specifications for any arrays they may use.

Working With Arrays

Panorama has about a dozen functions and procedure statements for working with arrays. These functions are described in this table.

Function	Reference Page	Description
array(text,item,sep)	Page 5036	<p>This function extracts a single data item from a text array. Text is the item of text that contains the data you want to extract. Item is the number of the data item you want to extract. The first item is item 1, the second is item 2, the third item is 3, etc. Separator is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the ␣ character.</p> <p>The array(function returns a single item of text from the array. Only the item itself is returned, the separator characters on each end are not included. If the item does not exist (for example if you ask for item 12 from a 7 item array) the function will return empty text (“”).</p> <p>There are 7 VHF television stations in Los Angeles. The example procedure below will convert channel numbers into the names of the stations. For example, the procedure converts Channel 7 into KABC.</p> <pre>Stations=" ,KCBS , ,KNBC ,KTLA , ,KABC , ,KCAL , ,KTTV , ,KCOP " «Channel Name»=array(Stations,7," , ")</pre> <p>The example uses an array called Stations. This array uses commas as a separator character.</p>
arrayboth(a1,a2,sep)		This statement compares two arrays. The result is a list of elements that are included in both arrays. Note: Empty array elements, if any, will be ignored. Both arrays must use the same separator.
arraybuild(sep,db,formula)		This function builds an array by scanning all records in a database. Note that the usage of this function is slightly different than the arraybuild statement in that the formula itself must be quoted. For example, instead of just upper(Name) you would need to use <code>"upper(Name)"</code> or <code>{upper(Name)}</code> .
arraychange(text,value,item,sep)	Page 5040	<p>This function changes a single value inside a text array. Only the one item is changed, all the other items in the array remain the same. Text is the text array that contains the data you want to change. Value is the new value of the data item. Item is the number of the data item you want to change. Items are numbered starting from 1 (1,2, 3,...). This item must already exist in the array. The arraychange(function will not add the item if it does not exist. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the ␣ character. This function returns a copy of the text array, with the data item changed. If you want to change the original array you should use an assignment statement (see below).</p> <p>The example procedure below will change the 5th item of the array to Navajo White.</p> <pre>Colors=arraychange(Colors,"Navajo White",5,";")</pre> <p>This example assumes that a field or variable named Colors already exists.</p>
arraycolumn(array,colnum, rowsep,colsep)		This function extracts a column from a two dimensional array. The second parameter specifies what column to extract. The third parameter specifies the main separator between each row of the array, the fourth parameter is the sub-separator between each column.

Function	Reference Page	Description
arraycontains(text,item,sep)		This function checks to see if any element of an array matches the specified text. For the result to be true, the array element must match the specified text exactly, including upper and lower case. Otherwise the function will return false. So checking for "Green" will only match that exact array element, not "green" or "Olive Green". (Note that this is quite different from the contains operator, which ignores upper and lower case and allows a submatch.)
arraydeduplicate(text,separator)		This function removes duplicate values from an array. As a byproduct it also sorts the array.
arraydelete(text,item,count,sep)	Page 5043	<p>This function deletes one or more elements from the middle of a text array. Text is the text array that you want to insert elements into. Item is the spot where you want the elements to be deleted. Count is the number of elements you want to delete from the array. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the ␣ character. This function returns a copy of the original text array, with the specified elements deleted from the middle. The example procedure below will delete the 3rd item from the SpeedDial array:</p> <pre style="text-align: center;">SpeedDial=arraydelete(SpeedDial),3,1,¶)</pre>
arraydeletevalue(text,value,sep)		This function deletes any array elements that match the value parameter. This must be an exact match, including upper and lower case. If the value occurs multiple times in the array, every occurrence of the value will be removed, with one exception: If the value occurs in two consecutive array elements, only the first occurrence will be deleted.
arraydifference(a1,a2,sep)		This statement compares two arrays. The result is a list of elements that are in the first array but not the second. (Note: Empty array elements, if any, will be ignored.) Both arrays must use the same separator.

Function	Reference Page	Description
arrayelement(text,position,sep)	Page 5044	<p>This function converts between character positions and array element numbers in a text array. Given a character position within the overall text, the arrayelement(function tells what array element the character is in. For example, in the array red;blue;green the 7th character (u) is in the 2nd array element.</p> <p>This function has three parameters: text, position and sep. Text is the text array that you are working with. Position is the position of the character within the overall text (starting with 1 for the first character). Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the ␣ character.</p> <p>This function returns a number. This is the number of data element in the array corresponding to the character position parameter. If the position corresponds to a separator character, the function will return the element number of the data element to the right of the separator.</p> <p>The example procedure below adds a new color to the RecentColors array. It then arbitrarily cuts off the array so that it is less than 200 characters long. The arrayelement(function makes it possible to write this procedure so that the array can be cut off without cutting an array element in the middle.</p> <pre> local lastElement RecentColors= parameter(1)+ sandwich(¶,RecentColors, " ") lastElement=arrayelement(RecentColors,200,¶) RecentColors= arrayrange(RecentColors,1,lastElement,¶) </pre> <p>This procedure could be useful for maintaining a pop-up menu of recently used colors. The procedure automatically keeps the menu to a reasonable size by lopping off old colors from the bottom if the array gets over 200 characters long.</p>
arrayfirst(text,sep)		This function extracts the first element of an array.
arrayinsert(text,item,count,sep)	Page 5047	<p>This function inserts one or more elements into the middle of a text array. Text is the text array that you want to insert elements into. Item is the spot where you want the new elements to be inserted. Count is the number of blank elements you want to insert into the array. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the ␣ character.</p> <p>This function returns a copy of the original text array, with the new blank array elements inserted into the middle. The example procedure below will add 5 new array items to the SpeedDial array between the 2nd and 3rd array items:</p> <pre> SpeedDial=arrayinsert(SpeedDial),¶,3,5) </pre> <p>The new array items created by arrayinsert(are blank (empty). You can fill them in with the arraychange(function.</p>
arraylast(text,sep)		This function extracts the last element of an array.
arraylefttrim(text,count,sep)		Removes the first elements of an array. For example arraylefttrim(text,2,") removes the first two elements from a comma separated array.

Function	Reference Page	Description
arraylookup(text,key,mainsep,subsep,default)		<p>This function looks up a value in a double column table, similar to a lookup. The first column in the table is the key value, the second column is the data value. Each line in the table is separated by the mainsep character, while the two columns in each line are separated by the subsep character. If no match is found the default value is returned. For example this function: <code>arraylookup("AL.ALABAMA;AK.ALASKA; ... WY.WYOMING",State,":",".", "")</code> can be used to look up a long state name given the two letter abbreviation.</p>
arraymerge(array1,array2,separator,joiner)		<p>This function merges two text arrays together. This function has four parameters: array1, array2, separator and joiner. Array1 is the first text array you want to merge, array2 is the second array. Separator is the separator character for both arrays (in other words, both arrays must use the same separator). This should be a single character. For carriage return delimited arrays, use the ¶ character (option-7). For tab delimited arrays use the ␣ character (option-L). Joiner is from 1 to 10 characters of text that will be used to join the individual elements of the two arrays.</p> <p>The result is a new array with the elements of the original arrays joined together. This means that the first element of the first array will be joined with the first element of the second array, then the second element of the second array will be joined with the second element of the second array, and so on until both arrays are completely merged.</p> <p>The example could be used to display names and phone numbers from a contact database in a Text Display SuperObject.</p> <pre>arraymerge(lookupall("Contacts" , "Company" , "Name" , ¶) , lookupall("Contacts" , "Company" , "Phone" , ¶) , ¶ , " , ")</pre> <p>The display will look something like the text shown below.</p> <p>John Smith , (510) 323-4905 Susan Wilson , (510) 590-1341 Bill Franklin , (510) 323-6781</p>
arraynotcontains(text,item,sep)		This function is the reverse of the arraycontains(function.
arraynumericssort(text,separator)		This function sorts an array in numerical order (instead of alphabetical order).
arraynumerictotal(text,separator)		This function totals the numerical elements of an array. For this function to work each array element must contain a number. The result is a number.
arrayrandomize(text,separator)		This function re-arranges the elements of an array in random order.

Function	Reference Page	Description
arrayrange(text,start,end,sep)	Page 5050	<p>This function extracts a series of data item from a text array. Text is the item of text that contains the data you want to extract. Start is the number of the first data item you want to extract. Items are numbered starting from 1 (1, 2, 3,...). End is the number of the last data item you want to extract. Items are numbered starting from 1 (1, 2, 3,...). Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the – character.</p> <p>This function returns a series of items from the array. It returns the first item, the last item, and everything in between (including any separators that are in between). If the last item does not exist (for example if you ask for item 12 from a 7 item array) the function will return up to the actual last item in the array. If both requested items do not exist, the function will return empty text (“”).</p> <p>This example procedure will fill the variable WeekDays with the text Mon,Tue,Wed,Thu,Fri.</p> <pre>Days="Sun,Mon,Tue,Wed,Thu,Fri,Sat" WeekDays=arrayrange(Days,2,6,"")</pre>
arrayreplacevalue(text,oldvalue,newvalue,sep)		<p>This function replaces any array elements that match the value parameter. This must be an exact match, including upper and lower case. If the value occurs multiple times in the array, every occurrence of the value will be replaced, with one exception. If the value occurs in two consecutive array elements, only the first occurrence will be replaced.</p>
arrayreverse(text,sep)	Page 5051	<p>This function reverses the order of the elements in a text array. In other words, the first element becomes the last element, the second element becomes the second to last, etc. Text is the text array that you want to modify. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “Special Characters” on page 57). For tab delimited arrays use the – character.</p> <p>The arrayreverse(function reverses the order of the elements of an array. For example, the formula:</p> <pre>arrayreverse("1;2;3;4",";")</pre> <p>will produce the array 4;3;2;1.</p>
arrayreverselookup(text,key,mainsep,subsep,default)		<p>This function looks up a value in a double column table, similar to a lookup. The first column in the table is the key value, the second column is the data value. However, this function reverses the function of these two columns. The key parameter is looked up in the second column, then the associated value is returned from the first column. This is the reverse of the arraylookup(function. Each line in the table is separated by the mainsep character, while the two columns in each line are separated by the subsep character. If no match is found the default value is returned. For example this function: arrayreverselookup("AL.ALABAMA;AK.ALASKA; ... WY.WYOMING",State,";",".",") can be used to look up a two letter state abbreviation given the full name of the state.</p>

Function	Reference Page	Description
arrayscan(field,sep)	Page 5052	<p>This function allows the individual elements of a text array in a database field to be exported on separate lines. Field is the name of the field that contains the array you want to export. (You can also use a variable, but this usually doesn't make sense). Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see "Special Characters" on page 57). For tab delimited arrays use the - character.</p> <p>This function returns one element from the array. However, unlike the array(function, the arrayscan(modifies the way the export and arraybuild statements work. These statements will repeat the formula containing arrayscan(over and over again for each record. Each time, the function will return the next element in the array, until there are no more items.</p>
arraysearch(array,text,start,sep)	Page 5054	<p>This function searches a text array to see if it contains a specific value. Array is the text array that you want to search. Text is the text that you want to search for. This parameter may contain the wildcard characters ? and * . For example, to search for array items that start with John use John* . To search for any array item containing Pacific use *Pacific* . The array item must match the text exactly, including upper/lower case. For more information on wildcard characters, see "A match B" on page 126. Start is the spot in the array where you want the search to begin from. If you want to search the entire array, this parameter should be one. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see "Special Characters" on page 57). For tab delimited arrays use the - character.</p> <p>If the arraysearch(function finds an array element that matches what you are searching for it returns the number of that array element (1, 2, 3, etc.). If there is no matching element, the function returns 0.</p>
arraysselectedbuild(sep,db,formula)		<p>This function builds an array by scanning the selected records in a database. Note that the usage of this function is slightly different than the arraysselectedbuild statement in that the formula itself must be quoted. For example, instead of just upper(Name) you would need to use "upper(Name)" or {upper(Name)}.</p>
arraysize(text,sep)	Page 5057	<p>This function counts the number of items in a text array. Text is the text array that you want to count. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see "Special Characters" on page 57). For tab delimited arrays use the - character.</p> <p>This function returns a number. This is the number of elements in the array. If there is no text in the array, the function will return one. If you need a function that returns zero if there is no text you can use the extract function with the last parameter set to -1 (see "String Modification Functions" on page 80).</p> <p>This example uses the arraysize(function to display the number of forms in the current database. (The dbinfo("forms","") function creates an array listing all the forms in the current database, separated by carriage returns.)</p> <pre>message "This database contains "+ str(arraysize(dbinfo("forms",""),¶))+" forms"</pre>
arraysort(text,separator)		<p>This function sorts an array in alphabetical order.</p>

Function	Reference Page	Description
arraystrip(text,sep)	Page 5059	This function removes any blank elements from a text array. Text is the text array that you want to strip the blank elements from. Sep is the separator character for this array. This should be a single character. For carriage return delimited arrays, use the ¶ character (see “ Special Characters ” on page 57). For tab delimited arrays use the ␣ character. This function returns a copy of the original text array, with any blank array elements removed from the array.
arraytableceiling(array,key,sep,subsep,default)		This function looks up a value in a double column table, similar to the table(function but from an array instead of a database. The table must be sorted in ascending order. The first column in the table is the key value, the second column is the data value. (Note: If the sub separator is "", the first column is used as the data value also.) If there is no exact match this function will return the next higher value, if any. If the key value is numeric or contains all numbers, numeric comparisons ($50 < 200$) will be used instead of text comparisons ($50 > 200$). (Note: This function always returns a text value, even if the key is numeric.) If the key value is not within the array bounds the default value will be returned.
arraytablefloor(array,key,sep,subsep,default)		This function looks up a value in a double column table, similar to the table(function but from an array instead of a database. The table must be sorted in ascending order. The first column in the table is the key value, the second column is the data value. (Note: If the sub separator is "", the first column is used as the data value also.) If there is no exact match this function will return the next lower value, if any. If the key value is numeric or contains all numbers, numeric comparisons ($50 < 200$) will be used instead of text comparisons ($50 > 200$). (Note: This function always returns a text value, even if the key is numeric.) If the key value is not within the array bounds the default value will be returned.
arraytrim(text,count,sep)		This function removes the last elements of an array. For example <code>arraytrim(text,2,",")</code> removes the last two elements from a comma separated array.
arrayunpropagate(text,separator)		This function scans an array from top to bottom. If it finds two or more duplicate values in a row, it blanks out all but the first. This is similar to Panorama's <code>UNPROPAGATE</code> command in the Math menu, but for an array instead of a database field.
lineitemarray(field,separator)	Page 5468	This function converts the data in a set of line item fields into a text array (see “ Text Arrays ” on page 93). Field is the line item field that contains the data. You should put the line item field name in quotes, and it should end with the Ω symbol (see “ Special Characters ” on page 57). Separator is the separator character for this array. This should be a single character. This function returns a copy of the line item data packed into an array. If the line items contain numbers or dates they are converted to text before being added to the array.
makenumberedarray(sep,start,end)		This function generates a numeric sequenced array, for example 1, 2, 3, 4, 5. You can specify the starting and ending number of the sequence.

HTML Tag and Tag Parsing Functions

Panorama has several functions for working with text that contains data delimited by tags. These functions are not actually specific to HTML, and you may find other uses for them.

These functions treat a tag as three components: **header**, **body** and **trailer**. In this example the tag header is `<`, the tag trailer is `>`, and the tag body is `IMG SRC="happy.gif"`.

```
<IMG SRC="happy.gif">
```

The tag header and trailer may be more than one character long. Here is the same tag but with only the picture name as the body. In this example the tag header is ``, and the tag body is `happy.gif`.

```
<IMG SRC="happy.gif">
```

The tag functions don't care about upper or lower case, so this tag will work fine if it is ``.

Here are descriptions of all of the tag parsing functions.

Function	Reference Page	Description
<code>htmltabletoarray(table, rowsep,colsep,columns)</code>		This function converts an HTML table into a two dimensional text array. You must specify two separator characters — the row separator (between lines) and the column separator (between columns). The fourth parameter is a list of table columns to include, comma separated. For example "2,5,6,10" would tell this function to include 4 of the HTML columns in the final array. If this parameter is set to "" then all of the columns in the HTML table will be included in the output array.
<code>tagarray(text,header,trailer,sep)</code>	Page 5830	<p>This function builds an array containing the body of all the tags in the text. The header is the character or sequence of characters that appears at the start of each tag. To extract all HTML tags the header would be <code><</code>. To extract all image tags the header would be <code><img</code> or <code><IMG</code> (either upper or lower case will work). The trailer is the character or sequence of characters that appears at the end of each tag. To extract all HTML tags the trailer would be <code>></code>. Each element in the array is separated from the next with the sep character. This character is often a carriage return (¶) or comma, see "Text Arrays" on page 93.</p> <p>This example displays a list of all HTML tags in the variable Page. The list will be separated by commas, for example <code>H1,/H1,B,/B,IMG SRC="my picture.jpeg"</code>.</p> <pre>message tagarray(Page,"<",">","")</pre> <p>This example lists all of the pictures in the variable Page.</p> <pre>message tagarray(Page,"","¶")</pre> <p>This will list each image on a separate line, like this:</p> <pre>src="happy.gif" src="rocket.jpeg" SRC="My Picture.jpeg" src="logo.gif"</pre>

Function	Reference Page	Description
tagcount(text,header,trailer)	Page 5832	<p>This function returns the number of tags in the text. The header is the character or sequence of characters that appears at the start of each tag. To count all HTML tags the header would be <code><</code>. To count all image tags the header would be <code><img</code> or <code><IMG</code> (either upper or lower case will work). The trailer is the character or sequence of characters that appears at the end of each tag. To count all HTML tags the trailer would be <code>></code>. Here is an example that uses this function to count the number of links in an HTML document.</p> <pre>message "This page contains "+ str(tagcount(Page,"<A HREF",">"))+" links."</pre>
tagdata(text,header,trailer,number)	Page 5833	<p>This function returns the body of the specified tag. The header is the character or sequence of characters that appears at the start of each tag. To extract an HTML tag the header would be <code><</code>. To extract an image tags the header would be <code><img</code> or <code><IMG</code> (either upper or lower case will work). The trailer is the character or sequence of characters that appears at the end of each tag. To extract an HTML tag the trailer would be <code>></code>. The number parameter specifies which tag you want: 1, 2, 3, etc. for the first, second, third tag etc.</p>
tagstart(text,header,trailer,number)	Page 5840	<p>These functions return the starting and ending position of the body of a tag within a text. The header is the character or sequence of characters that appears at the start of each tag. The trailer is the character or sequence of characters that appears at the end of each tag. The number parameter specifies which tag you want: 1, 2, 3, etc. for the first, second third tag etc. This example displays 20 characters or so around the first image tag.</p> <pre>message Page[tagstart(Page,"<IMG",">",1)-20, tagend(Page,"<IMG",">",1)+20]</pre> <p>If the requested tag does not exist, the tagstart(and tagend(function will return 0.</p>
tagend(text,header,trailer,number)	Page 5834	
tagnumber(text,header,trailer,pos)	Page 5835	<p>This function checks to see if position is in a tag within text, and if so, returns the number of that tag within the document (1, 2, 3). If the position is not inside of a tag the function will return zero. The header is the character or sequence of characters that appears at the start of each tag. The trailer is the character or sequence of characters that appears at the end of each tag.</p>
striphtmltags(text)		This function removes all HTML tags from the text.
xtag(tag,text)		This function generates an HTML/XML tag. For example the formula <code>xtag("italic","hello world")</code> will generate the text <code><italic>hello world</italic></code> .
xtagvalue(text,tag)		This function extracts the data from the first matching HTML or XML tag in the text. For example the formula <code>xtagvalue(page,"title")</code> will extract the page title from an HTML page.

Tag Parameter Functions

Many HTML tags contain parameters. For example, this tag has three parameters, **src**, **align** and **border**.

```
<IMG SRC="mylogo.gif" align=left border=0>
```

Panorama has built in functions that can help you extract a series of parameters like this. Although these functions were designed with parsing HTML tags in mind you may find other uses for them as well.

Function	Reference Page	Description
tagparameter(text,name,num)	Page 5837	<p>This function returns the value of a specified parameter in the tag. Text is the list of parameters. If you are parsing HTML this should be the body of the tag. Name is the name of the parameter you want to extract, including any trailing punctuation (= for HTML tags). Either upper or lower case is ok. For example to extract the name of the image itself from an image tag (see example above) the name would be src= or SRC=. To extract the alignment the name would be align= or ALIGN=. The num is in case there is more than one parameter with the specified name, it tells Panorama which one to extract (1, 2, 3, etc.)</p> <p>If the parameter value is quoted (for example src="my logo.jpg" Panorama will remove the quotes as it extracts the value. If the parameter value is not quoted it will extract up to the first non-alphanumeric value. For example, this formula will return the image file name of the first IMG tag in a field named HTML.</p> <pre>tagparameter(tag(HTML,"<img",">"),"src=",1)</pre> <p>If the first image tag in this text is this function will return mylogo.gif.</p>
tagparameterarray(text,name,sep)	Page 5838	<p>This function returns an array of the specified parameters in a tag. This is useful if the same parameter may occur multiple times within the tag. Text is the list of parameters. If you are parsing HTML this should be the body of the tag. Name is the name of the parameter you want to extract, including any trailing punctuation (= for HTML tags). Either upper or lower case is ok. For example to extract the name of the image itself from an image tag (see example above) you the name would be src= or SRC=. To extract the alignment the name would be align= or ALIGN=. Sep is the separator character that will be placed in between each value in the output array (see "Text Arrays" on page 93).</p> <p>To illustrate this function, suppose that you have a field named HTML that somewhere within it contains text that looks like this:</p> <pre><MERGE field="Name" field="Address" field="City" field="State"></pre> <p>Using this formula we can extract an array of all the field names.</p> <pre>tagparameterarray(tag(HTML,"<merge",">"),"field=",";")</pre> <p>With the sample data listed above this formula will return the value Name;Address;City;State.</p>

HTML Table Parsing Functions

These functions are specifically designed for extracting data from an HTML table.

Function	Reference Page	Description
htmltablecell(table,row,cell)		This function extracts the data from a cell in an HTML table. Any HTML tags in the cell are removed, leaving only the actual text. The thetable parameter must contain the body of an html table with <tr> and <td> tags (however, the actual <table> and </table> tags themselves are not required.)
htmltablecellexists(table,row,cell)		This function checks to see whether a cell in an HTML table exists or not. The result will be true if the cell exists, or false if it doesn't. The thetable parameter must contain the body of an html table with <tr> and <td> tags (however, the actual <table> and </table> tags themselves are not required.)
htmltablecellraw(table,row,cell)		This function extracts the data from a cell in an HTML table. Unlike the htmltablecell(function, any HTML tags in the cell are retained. The thetable parameter must contain the body of an html table with <tr> and <td> tags (however, the actual <table> and </table> tags themselves are not required.)
htmltableheight(table)		This function calculates the height (number of rows) in an HTML table. It assumes that the table is a regular matrix (no rowspan tags).
htmltablerowraw(table,row,cell)		This function extracts the data from a row in an HTML table. Any HTML tags in the row are retained. The the table parameter must contain the body of an html table with <tr> and <td> tags (however, the actual <table> and </table> tags themselves are not required.)
htmltablewidth(table)		This function calculates the height (number of rows) in an HTML table. It assumes that the table is a regular matrix (no colspan tags) and that all of the rows have the same number of columns as the top row in the table.

HTML/URL Conversion Functions

The HTML and URL standards used on the Internet do not use standard ASCII text. Panorama includes conversion functions for converting between standard ASCII and HTML and URL's. These functions are very convenient for generating HTML from a database, for example in CGI code for a web server.

Function	Reference Page	Description
htmlencode(text)	Page 5346	<p>This function converts from standard ASCII to HTML. Wherever possible, special characters are converted to their HTML equivalents (for example © is converted to &copy;, & is converted to &amp;. Special characters that do not have HTML equivalents are removed. (However, the smart quote characters, “,” ‘ and ’ are converted to regular quote characters " and '.)</p> <p>To allow you to convert HTML text that contains tags, the htmlencode(function does not convert the < and > characters. If you want to convert these characters into their HTML equivalents use this formula:</p> <pre>replacemultiple(htmlencode(text), "<.>", "&lt;.&gt;", ".")</pre>
htmldecode(text)	Page 5343	This function converts from HTML to standard ASCII, the exact opposite of the htmlencode(function. HTML characters like © and & are converted into the normal ASCII characters © and &.

Function	Reference Page	Description
urlencode(text)	Page 5884	The urlencode(function converts standard ASCII to URL format. For example, the text My URL would be converted to My%20URL .
urldecode(text)	Page 5883	The urldecode(function converts a URL to standard ASCII format. For example the text My%20URL would be converted to My URL .
urlfilename(url)		This function extracts the filename from a complete url.
urlpath(url)		This function extracts the path from a url.
webtext(text)		This function converts a number or regular text into text that may be displayed on a web browser. As far as possible, any special characters are converted into HTML entities so that they will display correctly (for example accented characters like à, special characters like ©, and the < and > symbols). This function goes beyond the htmlentities function in that it also encodes <, >, and carriage return (converted into).

HTML Generating Functions

These functions help with generating HTML.

Function	Reference Page	Description
htmlbold(string)		This function takes the text and adds and tags to it.
htmlitalic(string)		This function takes the text and adds <i> and </i> tags to it.
weblink(url,caption)		This function builds an HTML link tag. If the caption is not supplied ("" the URL will be used for the caption.
weblinknewwindow(url,caption)		This function builds an HTML link tag. The link will open the new page in a new window instead of in the same window. If the caption is not supplied ("" the URL will be used for the caption.
xtag(tag,text)		This function generates an HTML/XML tag. For example the formula <code>xtag("italic","hello world")</code> will generate the text <code><italic>hello world</italic></code> .

Encoding/Decoding Base64 Data

Base64 is an algorithm used to encode binary data into text. Base64 is widely used on the web and in e-mail. For more information on this encoding method see <http://en.wikipedia.org/wiki/Base64>.

Function	Reference Page	Description
encodebase64(data,linelength)		This function encodes text using the Base64 algorithm. The data parameter is the data to be encoded. LineLength is the maximum line length of the encoded text. A common value is around 70 characters per line - MIME data may contain an maximum of 76 characters per line.
decodebase64(data)		This function decodes text that has been encoded using Base64 encoding. It is the reverse of the encodebase64(function described below. If you first use encodebase64(then decodebase64(you will get back the original data.

Date Arithmetic

Formulas can perform several useful calculations on dates. For example, you can calculate the number of days between two dates, or you can add or subtract a certain number of days to a date. You can also convert a date to text using a wide variety of formats.

Usually we think of a date in terms of years, months, and days. Formulas, however, treat dates as a certain number of days—specifically, the number of days between that date and January 1, 4713 B.C., adjusted for the Gregorian calendar correction in October 1582. (The date 4713 B.C. is chosen for obscure astronomical reasons). For example, to a formula the date August 7, 1991 is day number 2,448,476.

Fortunately you should never have to worry about numbers like 2,448,476. The formula will automatically convert a date field into the number of days, perform the calculation, and then convert back into a regular date again.

Since formulas handle dates as numbers, you can use any numeric operator or function to manipulate dates. However it doesn't make much sense to take the square root of a date (although Panorama will let you). There are really only two numeric operations that make sense on dates—subtracting two dates to find the number of days in between and adding or subtracting a number of days to a date.

To calculate the number of days between two dates, just subtract one from the other. For example, the formula

```
«Ship Date»-«Order Date»
```

will calculate the number of days required to process an order.

To calculate an offset from a given date, just add the number to the date. For example the formula

```
«Ship Date»+30
```

calculates the normal due date 30 days after the ship date.

Today's Date

The `today()` function ([reference page 5862](#)) returns the number corresponding to today's date, allowing you to use today's date in a formula. For example, to calculate the age of an invoice use a formula like this.

```
today()-«Ship Date»
```

To calculate the due date for a library book, use the formula like this.

```
today()+14
```

This formula assumes that books are checked out for two weeks.

Converting Between Dates and Text

These functions allow you to convert a date into text, or text into a date. You should only use these functions if you want to store the result of a date calculation in a text field instead of a date field, or if you want to access a date that has been stored as text.

Note: Remember, formulas handle dates as numbers, so these functions actually convert numbers into text and vice versa. It's up to you to make sure that these numbers actually represent the correct dates.

Function	Reference Page	Description
completedatestr(number)		Convert a date to text, including the day of the week (for example Sunday, April 20th, 2003).
date(text)	Page 5146	This function converts a text string in a date format into the number representing that date. Use this function to include a constant date in your formula, for example <code>date("12/9/1979")</code> . You should also use this function to access dates that have been stored in text fields (but why are you doing that in the first place?). Several formats are supported, including <code>mm/dd/yy</code> , <code>mm/dd/yyyy</code> , <code>Month dd, yyyy</code> , and <code>Mon dd, yyyy</code> . Dates in the current week can be represented by the name of the day, for instance Tuesday or Fri . Dates in the previous or upcoming week can be represented by adding the words last or next , for example last friday or next wed .
datepattern(number,pattern)	Page 5148	This function converts a number representing a date into a formatted text string. The pattern parameter is an output pattern telling the function how to format the date. For more information on date output patterns, see " Date Output Patterns " on page 255 of the <i>Panorama Handbook</i> . Use the <code>datepattern(</code> function to store a date in a text field, or to display a formatted date in an auto-wrap text object or Text Display SuperObject. For example, the formula: <code>datepattern(«Ship Date», "Month ddnth, yyyy")</code> can be used to display the date an order was shipped in the format May 12th, 2003 .
datestr(number)		Convert a date to text using format <code>mm/dd/yy</code> (for example 4/20/03).
daystr(number)		Convert a date to the day of the week (for example Sunday).
eurodatestr(number)		Convert a date to text in European format (for example 20-APR-2003).
exportcell(field)	Page 5209	This function takes any database field and converts it to text, using the appropriate pattern if one has been defined in the design sheet. Field is the name of the field to be converted to text. The function always returns a text type data item. The power of the <code>exportcell(</code> function is that it does not require you to know what type of data you are exporting. It simply takes whatever kind of data is in the field (text, number, date, whatever) and converts it into text.
longdatestr(number)		Convert a date to text with format <code>Month ddnth, yyyy</code> (for example April 20th, 2003).
naturaldata(date)		This function converts a date to text in a natural format similar to how people would refer to the date, for example Today, Tue, Apr 4. If the date is more than 180 days in the past or is in the future the pattern <code>mm/dd/yy</code> is used. This is similar to how the Apple Finder displays dates..

Date Functions

These functions perform various calculations and conversions on date values. Unless specified otherwise the date is always processed as a numeric value (see “[HTML Generating Functions](#)” on page 105).

Function	Reference Page	Description
datevalue(year,month,day)		This function converts three integers into a date. The three integers are the year, month and date. This function provides a way to create a date that is independent of the system date settings (the date(function, which can also create dates, will produce different values in different countries depending on the date formats used in those countries).
dayofweek(date)	Page 5149	<p>This function computes the day of the week (0-6) of a date, with Sunday being 0, Monday 1, etc. The function returns a number from 0 to 6. The days of the week are:</p> <ul style="list-style-type: none"> 0 Sunday 1 Monday 2 Tuesday 3 Wednesday 4 Thursday 5 Friday 6 Saturday <p>The procedure below uses the dayofweek(function to select all weekday records (monday through friday).</p> <pre>select dayofweek(Date)≥1 and dayofweek(Date)≤5</pre>
dayvalue(date)		This function extracts the day of the month from a date as a numeric value (1 to 31).
month1st(date)	Page 5532	<p>This function computes the first day of a month. For example, if the date passed to this function is October 18, 1997, this function will return the date October 1, 1997. The date is returned as a number.</p> <p>The example procedure below uses this function to select the orders placed this month, then displays the count.</p> <pre>select OrderDate≥month1st(today()) and OrderDate<month1st(today()+monthlength(today() message str(info("records"))+" orders this month"</pre>
monthlength(date)	Page 5533	<p>This function computes the length (number of days) of a month. For example, if the date passed to this function is October 18, 1997, this function will return 31, the number of days in October. This function knows about leap years and adjusts the length of February accordingly.</p> <p>The example procedure below uses this function to select the orders placed this month, then displays the count.</p> <pre>select OrderDate≥month1st(today()) and OrderDate<month1st(today()+monthlength(today() message str(info("records"))+" orders this month"</pre>

Function	Reference Page	Description
monthmath(date,offset)	Page 5534	<p>This function takes a date and computes another date that is one or months before or after the original date. Date is a number representing the original date. Offset is the number of months that you want to add or subtract to the original date. Use a positive number to move forward in time, a negative number to go backwards. For example, if you offset the date May 12, 1997 by 2 (two months forward) the result is July 12, 1997. If you offset the same original date by -2 (two months backward) the result is March 12, 1997.</p> <p>If the new date does not exist because a month does not have enough days in it, the monthmath(function will pick the last day of the month. For example, if you offset March 31 by 1 month the result is April 30. If the new month lands in February the function knows about leap years and adjusts accordingly.</p> <p>This example calculates a renewal date exactly one year from today.</p> <pre>monthmath(today(),12)</pre>
monthvalue(date)		This function extracts the month from a date as a numeric value (1 to 12).
quarter1st(date)	Page 5623	This function computes the first day of a quarter. For example, if the date passed to this function is August 18, 1997 , this function will return the date July 1, 1997 . The date is returned as a number.
quartervalue(date)		This function extracts the quarter within a year from a date as a numeric value (1 to 4).
today()	Page 5862	This function returns today's date (assuming, of course, that your computer clock has been set correctly).
week1st(date)	Page 5890	This function computes the first day of a week (Sunday). For example, if the date passed to this function is July 12, 1995 (a Wednesday), this function will return the date July 9, 1995 (a Sunday). The date is returned as a number.
year1st(date)	Page 5912	<p>This function computes the first day of a year. For example, if the date passed to this function is July 12, 1995, this function will return the date January 1, 1995. The date is returned as a number.</p> <p>The example below calculates the number of days remaining in the current year.</p> <pre>yearfirst(year1st(today())+366)-today()</pre>
weekvalue(date)		This function extracts the week from a date as a numeric value (this is the number of weeks since the start of the year, 1 to 52).
yearvalue(date)		This function extracts the year from a date as a numeric value.

Calendar Functions

The functions described in this section facilitate the creation of monthly calendars like this.

use calendarday(function to determine number for each box

August		1999				
Sun	Mon	Tue	Wed	Thu	Fri	Sat
1 Morning Service	2	3	4	5 Worship Practice	6	7
8 Morning Service	9	10	11	12 Worship Practice	13	14
15 Morning Service	16	17	18	19 Worship Practice	20	21
22 Morning Service	23	24	25	26 Worship Practice	27	28
29 Morning Service ProYUE 98	30 ProYUE 98 First Day Home/c	31 ProYUE 98 Second Day Cross Platform				

use calendardate(function to determine date value for each box

See “[Building a Calendar](#)” on page 971 of the *Panorama Handbook* for step-by-step instructions on building a calendar like this.

Function	Reference Page	Description																																																	
calendarday(date,boxnumber)	Page 5084	<p>This function is designed to help in creating monthly calendars. A standard monthly calendar has 6 rows and 7 columns (Sunday through Saturday) for a total of 42 boxes. For any given month from 28 to 31 of these boxes will be valid dates. The calendarday(function calculates what number from 1 to 31 (if any) should be displayed in one of these 42 boxes.</p> <p>This function has two parameters: date and boxnumber. date is any date in the month being displayed. Boxnumber is the box within the monthly calendar being displayed. The boxes are numbered from 1 to 42, starting with the upper left hand corner. The table below shows the position of all 42 monthly calendar boxes.</p> <table style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>S</th> <th>M</th> <th>T</th> <th>W</th> <th>T</th> <th>F</th> <th>S</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> <td>6</td> <td>7</td> </tr> <tr> <td>8</td> <td>9</td> <td>10</td> <td>11</td> <td>12</td> <td>13</td> <td>14</td> </tr> <tr> <td>15</td> <td>16</td> <td>17</td> <td>18</td> <td>19</td> <td>20</td> <td>21</td> </tr> <tr> <td>22</td> <td>23</td> <td>24</td> <td>25</td> <td>26</td> <td>27</td> <td>28</td> </tr> <tr> <td>29</td> <td>30</td> <td>31</td> <td>32</td> <td>33</td> <td>34</td> <td>35</td> </tr> <tr> <td>36</td> <td>37</td> <td>38</td> <td>39</td> <td>40</td> <td>41</td> <td>42</td> </tr> </tbody> </table> <p>This function returns a number from 1 to 31, or zero if the specified calendar box does not contain a day in this month.</p>	S	M	T	W	T	F	S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
S	M	T	W	T	F	S																																													
1	2	3	4	5	6	7																																													
8	9	10	11	12	13	14																																													
15	16	17	18	19	20	21																																													
22	23	24	25	26	27	28																																													
29	30	31	32	33	34	35																																													
36	37	38	39	40	41	42																																													

Function	Reference Page	Description																																																	
calendardate(date,boxnumber)	Page 5082	<p>This function is designed to help in creating monthly calendars. A standard monthly calendar has 6 rows and 7 columns (Sunday through Saturday) for a total of 42 boxes. For any given month from 28 to 31 of these boxes will be valid dates. The calendarday(function calculates what date corresponds to one of these 42 boxes.</p> <p>This function has two parameters: date and boxnumber. Date is any date in the month being displayed. Boxnumber is the box within the monthly calendar being displayed. The boxes are numbered from 1 to 42, starting with the upper left hand corner. The table below shows the position of all 42 monthly calendar boxes.</p> <table border="1" data-bbox="1006 723 1399 1006"> <thead> <tr> <th>S</th> <th>M</th> <th>T</th> <th>W</th> <th>T</th> <th>F</th> <th>S</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> <td>6</td> <td>7</td> </tr> <tr> <td>8</td> <td>9</td> <td>10</td> <td>11</td> <td>12</td> <td>13</td> <td>14</td> </tr> <tr> <td>15</td> <td>16</td> <td>17</td> <td>18</td> <td>19</td> <td>20</td> <td>21</td> </tr> <tr> <td>22</td> <td>23</td> <td>24</td> <td>25</td> <td>26</td> <td>27</td> <td>28</td> </tr> <tr> <td>29</td> <td>30</td> <td>31</td> <td>32</td> <td>33</td> <td>34</td> <td>35</td> </tr> <tr> <td>36</td> <td>37</td> <td>38</td> <td>39</td> <td>40</td> <td>41</td> <td>42</td> </tr> </tbody> </table> <p>This function returns a date value (a number), or zero if the specified calendar box does not contain a day in this month.</p> <p>The output of the calendardate(function is usually fed into a lookupall(, lookupcalendar(, or lookupptime(function. The last two functions can be used to lookup the events (appointments, to-do's, etc.) that occur on a particular day.</p>	S	M	T	W	T	F	S	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42
S	M	T	W	T	F	S																																													
1	2	3	4	5	6	7																																													
8	9	10	11	12	13	14																																													
15	16	17	18	19	20	21																																													
22	23	24	25	26	27	28																																													
29	30	31	32	33	34	35																																													
36	37	38	39	40	41	42																																													
lookupcalendar(file, reminderField, date, dataField, separator)	Page 5504	<p>This function builds a text array containing one item for every record in the target database where the date in the reminderField matches the date. Each item in the text array contains the value extracted from the dataField for that record.</p> <p>This function has five parameters: file, reminderField, date, dataField and separator. File is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use <code>info("databasename")</code>. ReminderField is the name of the field that you want to search in. This field must contain valid reminders (see "Reminders" on page 119). The field must be in the database specified by the first parameter. Date is the actual date that you want to match. For example if you want to look up all appointments on July 23rd, this should be <code>date("July 23")</code>. This parameter is often a field in the current database. DataField is the name of the field that you want to retrieve data from. For example if you want to retrieve appointment information, this should be the name of the field that contains that information. This must be a field in the database specified by the first parameter. Separator is the separator character for the text array you are building (see "Text Arrays" on page 93).</p> <p>The function returns a text array from all the records in the specified database where the reminderField and date match. This example returns a list of today's reminders.</p> <pre data-bbox="991 2231 1952 2270">lookupcalendar("Reminders",When,today(),Message,¶)</pre>																																																	

Function	Reference Page	Description
<p>lookupptime(file, reminderField, date, pattern, separator)</p>	<p>Page 5509</p>	<p>This function builds a text array containing one item for every record in the target database where the date in the reminderField matches the date. Each item in the text array contains the time of the corresponding reminder.</p> <p>This function has five parameters: file, reminderField, date, pattern and separator. File is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use <code>info("databasename")</code>. ReminderField is the name of the field that you want to search in. This field must contain valid reminders (see “Reminders” on page 119). The field must be in the database specified by the first parameter. Date is the actual date that you want to match. For example if you want to look up all appointments on july 23rd, this should be <code>date("July 23")</code>. This parameter is often a field in the current database. Pattern is the pattern you want to use to format the time. See the <code>timepattern()</code> function for details (reference page 5861). Separator is the separator character for the text array you are building (see “Text Arrays” on page 93).</p> <p>The function returns a text array from all the records where the reminderField and date match. The text array contains the times for the reminders that match.</p>
<p>lookuprtypes(file, reminderField, date, pattern, separator)</p>	<p>Page 5511</p>	<p>The <code>lookuprtypes()</code> function builds a text array containing one item for every record in the target database where the date in the reminderField matches the date. Each item in the text array contains the type of the corresponding reminder, either "a" (appointment) or "t" (to-do).</p> <p>This function has five parameters: file, reminderField, date, pattern and separator. File is the name of the database that you want to search and grab data from. The database must be open. If you want to search and grab from the current database, use <code>info("databasename")</code>. ReminderField is the name of the field that you want to search in. This field must contain valid reminders (see “Reminders” on page 119). The field must be in the database specified by the first parameter. Date is the actual date that you want to match. For example if you want to look up all appointments on july 23rd, this should be <code>date("July 23")</code>. This parameter is often a field in the current database. Pattern is not used, and should be "". Separator is the separator character for the text array you are building (see “Text Arrays” on page 93).</p> <p>The function returns a text array from all the records where the reminderField and date match. Each element of the text array contains either "a" or "t" for the reminders that match.</p>

Time Arithmetic

To Panorama, time is not hours, minutes, and seconds, but simply seconds. To be precise, a time is the number of seconds since midnight. For example, the time **4:32 AM** is **16,320** seconds after midnight. As you can see, a Panorama time is really a number in disguise. Since times are numbers, it's easy to compare them, sort them, or find the difference between them (number of seconds).

Converting Between Times and Text

Unlike dates, Panorama does not automatically provide a time data type that automatically converts a date in text format into a number. You must use a function to convert time in text format into seconds before you can do math calculations with the time, and use another function to convert back.

Function	Reference Page	Description								
now()	Page 5548	This function returns the current time (number of seconds since midnight). Of course the clock on your computer must be set correctly!								
seconds(text)	Page 5709	<p>This function converts text into a number representing a time. The function has one parameter — the text that you want to convert to a number representing a time. If the text includes an AM or PM suffix, the number of seconds is calculated from midnight (12 A.M.), otherwise it is calculated from 0:00:00 (elapsed time). The text must contain a valid time. Here are some examples of valid times:</p> <pre>4:13 PM 11:00 AM 2:30 18:45</pre> <p>This function returns a number representing the time. The number is the number of seconds since midnight. For example, if the time is 10:23 AM this function will return the number 37,380.</p>								
timepattern(number,pattern)	Page 5861	<p>This function converts a number representing a time into text. The function uses a pattern to control how the date is formatted.</p> <p>The function has two parameters: number and pattern. Number is the number that you want to convert to text. This number must be the number of seconds since midnight. Pattern is text that contains a pattern for formatting the date. The pattern is assembled from four components: hh (hours), mm (minutes) ss (seconds), and am/pm. Some of the more common time patterns are listed here:</p> <table border="1"> <thead> <tr> <th>Pattern</th> <th>Converted Text</th> </tr> </thead> <tbody> <tr> <td>"hh:mm:ss am/pm"</td> <td>4:32:17 pm</td> </tr> <tr> <td>"hh:mm am/pm"</td> <td>4:32 pm</td> </tr> <tr> <td>"hh:mm:ss"</td> <td>16:32:17</td> </tr> </tbody> </table> <p>If am/pm is left off the pattern the time will be formatted in 24 hour format, as shown on the last line of the table above. You should also leave off am/pm for converting elapsed times.</p>	Pattern	Converted Text	"hh:mm:ss am/pm"	4:32:17 pm	"hh:mm am/pm"	4:32 pm	"hh:mm:ss"	16:32:17
Pattern	Converted Text									
"hh:mm:ss am/pm"	4:32:17 pm									
"hh:mm am/pm"	4:32 pm									
"hh:mm:ss"	16:32:17									

Function	Reference Page	Description
time(text)	Page 5855	<p>This function converts text into a number representing a time. The function has one parameter — the text that you want to convert to a number representing a time. The time function allows you to leave out the colons in the time, and also allows you to leave off the am/pm. Here are some examples of valid times:</p> <pre> 4:13 PM 11:00 AM 2:30 18:45 230 4p midnight noon afternoon evening night nite </pre> <p>The time(function is very lenient about the format you use to enter the time. It will accept a time without colons, for example 425 pm instead of 4:25 pm. If there is no am or pm the time function will try to make an intelligent guess. For example, 230 is almost certainly 2:30 pm, not 2:30 am. By default, the time(function assumes that any time from 6:00 to 11:59 is AM, and any time from 12:00 to 5:59 is PM, but you can change these assumptions with the timedefaults statement (reference page 5858).</p> <p>The time(function will also convert “named” times: noon, midnight, morning, afternoon, evening, and night. This function assumes that morning is 9:00 am, afternoon is 1:00 pm, evening is 6:00 pm, and night is 10:00 pm. These assumptions can be changed with the timedefaults statement (reference page 5858).</p>
timestr(number)		Convert a number to text in am/pm time format (for example 9:34 AM).

Time Calculations

Once time has been converted into seconds you can perform arithmetic on it. For example, to calculate the number of hours worked from a time card use a formula like this (this formula assumes that **In** and **Out** are text fields containing times).

```
(seconds(Out)-seconds(In))/3600
```

(The division by 3600 converts the result into hours.)

To find out when a task will be finished that takes 2 1/2 hours to complete, use the formula

```
seconds(«Start Time»)+seconds("2:30")
```

Simple addition and subtraction does not compensate for time wrapping around midnight. For example, if you want to calculate the length of a shift that begins at 11 P.M. and ends at 7 A.M., you must add 24 hours to 7AM before subtracting the times. To solve this problem you can use one of the functions described below, or you can use a SuperDate, which combines time and date into a single number (see “[SuperDates \(combined date and time\)](#)” on page 118).

Function	Reference Page	Description
time24(time)	Page 5857	<p>This function takes a time and makes sure it falls within a 24 hour period. If the time is less than 24 hours, it is unchanged. If the time is greater than 24 hours, it is converted to the equivalent time in a 24 hour period (for example 30:00:00 is converted to 6:00:00).</p> <p>The time24(function can help with calculations of an ending time from a start time and duration. The basic formula for such a calculation is shown here.</p> $\text{EndTime}=\text{StartTime}+\text{Duration}$ <p>This formula works fine unless the interval extends over midnight. The time24(function adjusts the result to make sure it starts over at zero as it crosses midnight.</p> $\text{EndTime}=\text{time24}(\text{StartTime}+\text{Duration})$ <p>This formula will correctly calculate that 10:30 PM + 4 hours is 2:30 AM.</p>
timedifference(start,end)	Page 5859	<p>This function calculates the difference between two times. It works correctly even if the interval between the two times crosses over midnight. This function returns a time interval between -12 and +12 hours. See also the timeinterval(function, which returns a time interval between 0 and 24 hours.</p> <p>There are two parameters, start and end. Start is a number (number of seconds) representing the starting point of the time interval. End is a number (number of seconds) representing the ending point of the time interval. This function returns the number of seconds between the two times. For example, if the start time is 9:30 PM and the end time is 2:05 AM, the difference would be 4:35. But if the parameters are reversed and the start is 2:05 AM and the end is 9:30 PM, the difference is -4:35. If the result is positive, the end is after the start. But if the result is negative, the start is after the end.</p>
timeinterval(start,end)	Page 5860	<p>This function calculates the time interval between two times. It works correctly even if the interval between the two times crosses over midnight. This function returns a time interval between 0 and 24 hours. See also the timedifference(function, which returns a time interval between -12 and +12 hours.</p> <p>There are two parameters, start and end. Start is a number (number of seconds) representing the starting point of the time interval. End is a number (number of seconds) representing the ending point of the time interval. This function returns the number of seconds between the two times. For example, if the start time is 9:30 PM and the end time is 2:05 AM, the interval would be 4:35. But if the parameters are reversed and the start time is 2:05 AM and the end time is 9:30 PM, the interval is 19:25.</p>

Time Calculations with Text

Unlike the functions in the previous sections, these functions operate with time values in strings. There aren't as many functions available as for times expressed as numbers, but if your input and output values will be in strings using these function saves the intermediate conversion steps.

Function	Reference Page	Description
<code>texttimedifference(start,end)</code>		<p>This function calculates the difference between two times. Instead of being expressed as numbers, the input output times are expressed as text (for example 12:45 pm). This function works correctly even if the interval between the two times crosses over midnight. This function returns a time interval between -12 and +12 hours. See also the <code>timeinterval()</code> function, which returns a time interval between 0 and 24 hours.</p> <p>There are two parameters, start and end. Start is a string representing the starting point of the time interval. End is a string representing the ending point of the time interval. This function returns the time difference between the start and end. For example, if the start time is 9:30 PM and the end time is 2:05 AM, the difference would be 4:35. But if the parameters are reversed and the start is 2:05 AM and the end is 9:30 PM, the difference is -4:35. If the result is positive, the end is after the start. But if the result is negative, the start is after the end.</p>
<code>texttimeinterval(start,end)</code>		<p>This function calculates the time interval between two times. It works correctly even if the interval between the two times crosses over midnight. This function returns a time interval between 0 and 24 hours. See also the <code>timedifference()</code> function, which returns a time interval between -12 and +12 hours.</p> <p>There are two parameters, start and end. Start is a string representing the starting point of the time interval. End is a string representing the ending point of the time interval. This function returns the time between the start and end. For example, if the start time is 9:30 PM and the end time is 2:05 AM, the interval would be 4:35. But if the parameters are reversed and the start time is 2:05 AM and the end time is 9:30 PM, the interval is 19:25.</p>

Calculating Time Intervals Smaller Than One Second

The `info("tickcount")` function can be used to calculate time intervals down to 1/60th of a second (0.0165 second). This function returns the number of 1/60th second intervals since the computer was turned on. Here is an example that uses this function to delay for 1/4 second.

```
local beginDelay
beginDelay=info("tickcount")
loop
  nop
while info("tickcount")<beginDelay+15
```

This loop will delay for 1/4 second (plus or minus 1/60th second) on any computer, no matter what the processor speed.

Time Code Calculations (Video/Film)

Panorama user and visual effects supervisor Chris Watts has built an asset management system that he has used in the production of several major motion pictures, including **Pleasantville** and **300**. In the process Chris has developed about a dozen Panorama functions that are useful in performing calculations with timecodes used in video and film, which he has donated so that other Panorama users can take advantage of them. Here at ProVUE we're not video/film editing experts, so these functions are provided as-is with Panorama.

Function	Reference Page	Description
feetandframes(frames)		This function takes an integer and returns a feet and frame count for 35mm film.
kcadd(keycode,offset)		This function takes a keycode for 35mm film and adds a number of frames to it and returns a keycode as a string.
kcdiff(in,out)		This function calculates the number of frames between two frames of 35mm film, using inclusive counts.
kframes(feetplusframes)		This function returns an integer based on a feet and frames count for 35mm film.
kcoutfromlength(key,offset)		This function answers the question "What will the last frame be if I start my cut at the incode and my shot is some number of frames long"?
outcode(textcode,framerate)		This function adds one frame to a timecode. Works with 24, 25, or 30 frame code. This is useful for making an EDL, when you need the timecode reflected in the EDL to be one frame after the last frame of picture.
tc24to30(timecode)		This function converts 24 frame timecode to 30 frame non-drop timecode.
tc30to24(timecode)		This function converts 30 frame non-drop timecode to 24 frame timecode.
tcadd(textcode,offset,framerate)		This function accepts a timecode, as a string, and adds some number of frames, returns a timecode string. Works with 24, 25, and 30fps timecode.
tcdiff(in,out,framerate,edl)		This calculates the number of frames between two timecodes. Use 1 for outcodetype if you are copying the put code from an EDL, otherwise use 0 if you are using 'inclusive' timecode. If you don't know what I am talking about, call a film editor!
tcfames(timecode,framerate)		This function returns a number of frames from zero of a 24fps, 25fps, or 30fps timecode. This works for non-drop timecode only!
timecode(frames,framerate)		This function returns a timecode string corresponding to an integer. This works for non-drop timecode only.

SuperDates (combined date and time)

SuperDates combine the date and time into a single number...the number of seconds since January 1, 1904. SuperDates make it easy to calculate time intervals across multiple days. However, SuperDates take up more storage than regular dates, and are not as easy to work with. In addition, SuperDates are limited to dates between 1904 and 2040. Panorama has several functions for working with SuperDates.

Function	Reference Page	Description
superdate(date,time)	Page 5822	This function converts a regular date and a regular time into a superdate. The date parameter is a regular Panorama date (see " Converting Between Dates and Text " on page 107). This date must be between 1904 and 2040 A.D. The time parameter is the number of seconds since midnight, usually computed with the seconds(or time(functions (see " Converting Between Times and Text " on page 113). The result is a single value that combines both the date and time into a single number that can be used for multi-day calculations.
regulardate(number)	Page 5643	This function extracts a regular date (number of days from January 1, 4713 B.C.) from a superdate. You can then format this date with the datepattern(function (see " Converting Between Dates and Text " on page 107), as is shown in this example (which assumes that the field or variable Arrival contains a SuperDate value). <code>datepattern(regulardate(Arrival),"Month dd, yyyy")</code>
regulartime(number)	Page 5644	This function extracts a regular Panorama time (seconds since midnight) from a SuperDate. You can then format this date with the timepattern(function (see " Converting Between Times and Text " on page 113), as is shown in this example (which assumes that the field or variable Arrival contains a SuperDate value). <code>timepattern(regulartime(Arrival),"hh:mm am/pm")</code>
superdatepattern(number, datepattern,timepattern)		Converts a number containing a superdate to text, allowing you to specify the patterns for both the date and the time portions. For example the formula <code>superdatepattern(supernow(),"Month ddnth yyyy @ ","hh:mm am/pm")</code> will result in something like July 10th, 2008 @ 1:36 PM .
superdatessecondsstr(number)		Converts a number containing a superdate to text in standard format including (for example 4/20/03 9:56:37 AM).
superdatestr(number)		Converts a number containing a superdate to text in standard format (for example 4/20/03 9:56 AM).
supernow()		This function returns the number representing the current date and time as a superdate.

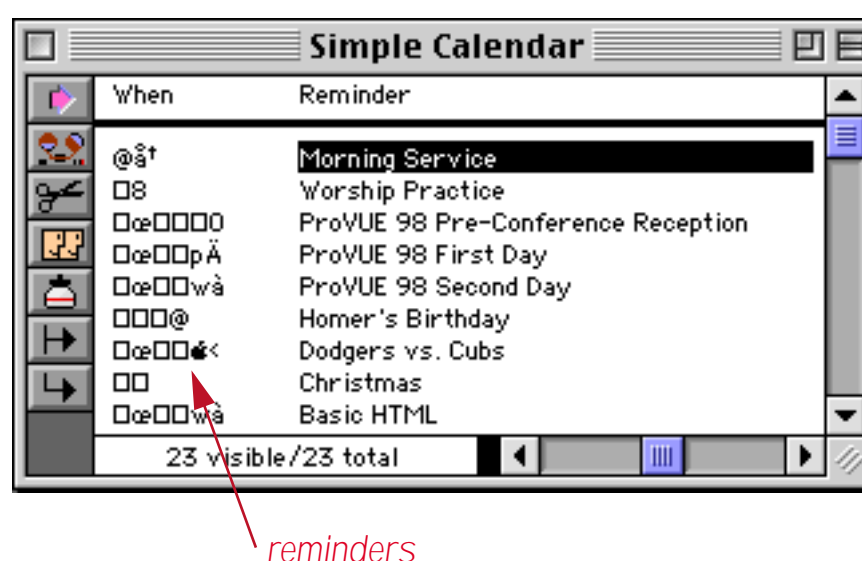
Reminders

A reminder is a special data type that holds scheduling information. Reminders are usually used in calendar database applications. A reminder is a raw binary data item 30 bytes long (stored in a text field or variable) and contains the following information:

- The reminder type (either appointment or to-do)
- The reminder date, or recurring date information (july 12, every tuesday, etc.)
- The reminder time (3:30pm, 7:20am, etc.)
- Alarm status
- Completion status (to-do only)
- Priority (to-do only)

Notice that the reminder only contains scheduling information. It does not contain any message or other information about the event. If there is a message associated with a reminder (for example, [Lunch with Bob](#)) it should be stored in a separate field or variable.

Although reminders can be kept in a variable, they are usually kept in a database field. Here's what reminder data looks like in the data sheet.



As you can see, reminders don't look like much in the data sheet. They are usually displayed in a form using the [remindercaption\(\)](#) function (see "[Reminder Functions](#)" on page 122).

Message	Date & Time
Morning Service	Sun of each week @10:00 AM
Worship Practice	Thu of each week @7:30 PM
ProVUE 98 Pre-Conference Reception	Sunday, August 29th, 1999 @7:00 PM
ProVUE 98 First Day	Monday, August 30th, 1999 @8:00 AM
ProVUE 98 Second Day	Tuesday, August 31st, 1999 @8:30 AM
Homer's Birthday	August 30th of each year @8:00 PM
Dodgers vs. Cubs	Sunday, August 29th, 1999 @5:05 PM
Christmas	December 25th of each year @12:00 AM
Basic HTML	Monday, August 30th, 1999 @8:30 AM

Reminders can also be displayed in a monthly calendar format. See "[Building a Calendar](#)" on page 971 of the *Panorama Handbook* for step by step instructions on setting up this format using a Super Matrix object.

Appointments vs. To-Do's

There are two different types of reminders: Appointments and To-do's. **Appointment** reminders are used for anything that has a definite, fixed, time: appointments, birthdays, meetings, etc. Once the time has passed the appointment is no longer relevant. For example, it won't do much good to be reminded that your spouse's birthday was yesterday!

To-do reminders have a completion status as well as a time and date. For example, suppose you set up a to-do reminder to order parts on Monday. If you don't get around to it, you'll still want it on your to-do list on Tuesday, and again on Wednesday etc. until you actually do order the parts. To-do reminders remain active until the task is completed (or at least it is marked as completed!)

Creating and Modifying a Reminder

Although reminders can be in a variable, they are usually kept in a database field. For the following example, we are going to assume that we have a database that contains fields named **Reminder** and **Message**. Both of these must be text fields.

To create a new reminder, you'll need to add a new record to the database, then use the **buildreminder** statement to allow the user to edit the reminder (see "**BUILDREMINDER**" on page 5078 of the *Panorama Reference*). Here's a procedure to do that (see "**Writing a Procedure from Scratch**" on page 216).

```
addrecord
buildreminder today(),now(),0,Reminder
reminder Reminder,Message
```

The **buildreminder** statement creates a new reminder. This statement has four parameters:

```
buildreminder date,time,type,field
```

The first parameter, **date**, is the date for this reminder. Since usually you are going to have the user edit the reminder right away, this is just a default date to get them started. In the example above we used today's date.

The second parameter, **time**, is the time for this reminder. Again, this is usually just a default until the user edits the time. In the example above we used the current time.

The third parameter, **type**, specifies whether this is an appointment (0) or a to-do reminder (1).

The fourth parameter, **field**, specifies what field the new reminder should be placed into.

Another way to create a new reminder is with the **reminder()** function (see "**REMINDER()**" on page 5648 of the *Panorama Reference*). This is similar to the **buildreminder** statement, but with an important difference. The function has three parameters:

```
reminder(date,time,type)
```

The first parameter specifies the **date** for the new reminder. However, do not use the date data type here. This function wants to see text that describes the date, for example "2/7/96", "july 3", or "last tuesday".

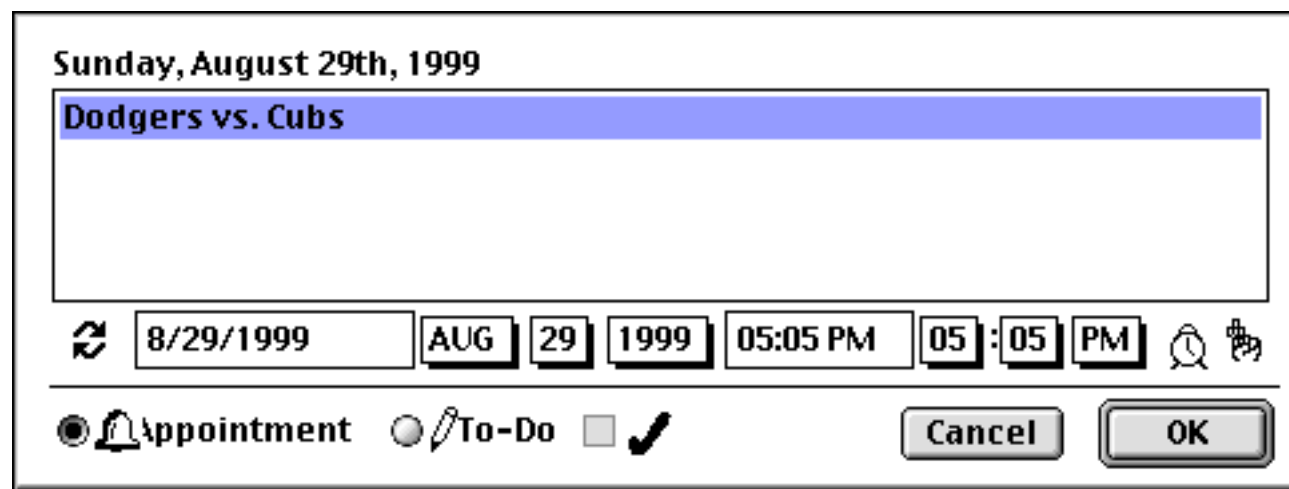
The second parameter specifies the **time** for the new reminder. However, do not use a number here. This function wants to see text that describes the time, for example "5:20 pm".

The third parameter specifies whether this new reminder should be an appointment ("a") or a to-do reminder ("t").

Here is our example rewritten to use the **reminder()** function:

```
addrecord
Reminder=reminder("today","12:00 pm","a")
reminder Reminder,Message
```

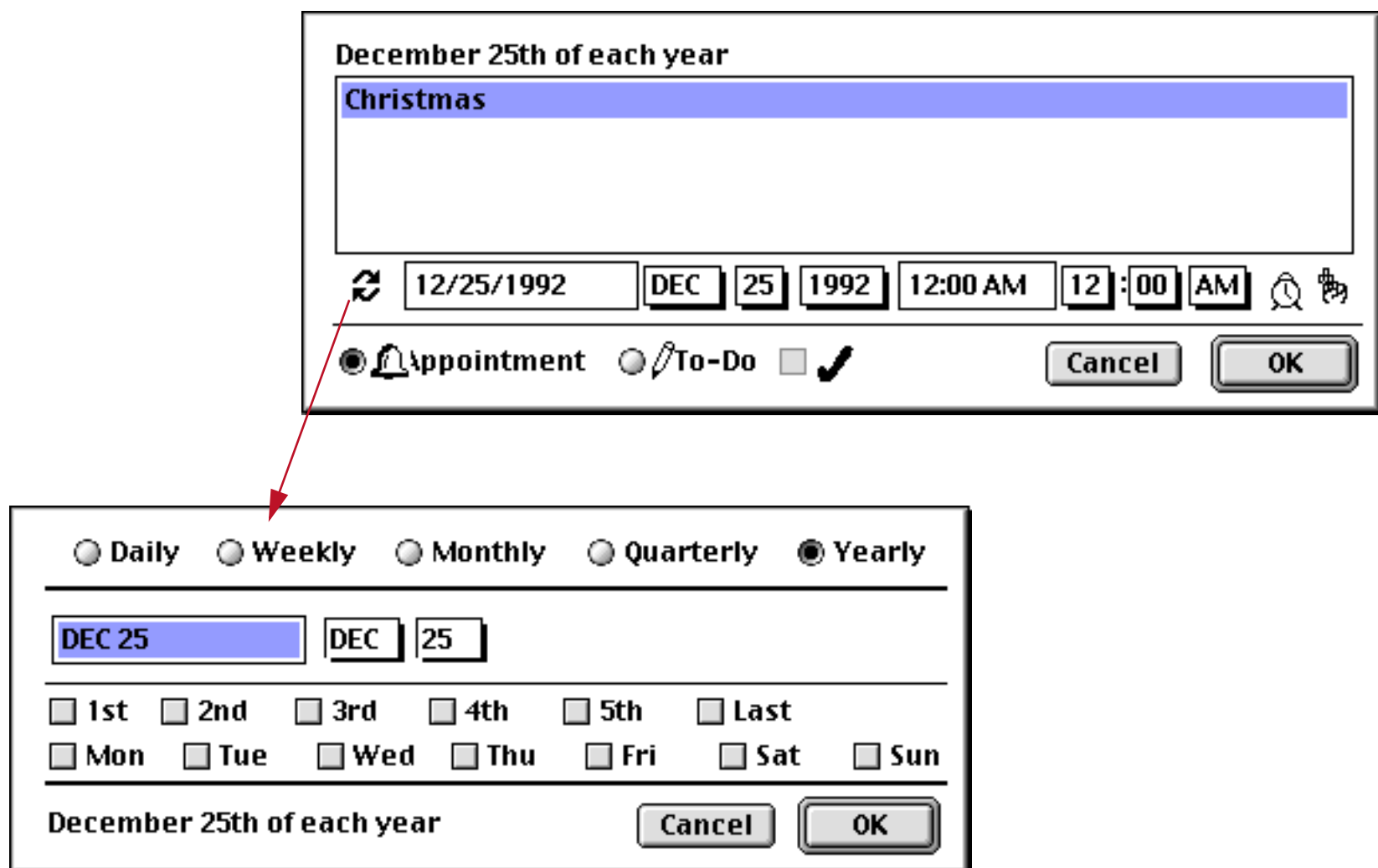

The `reminder` statement is used to edit reminders. This statement displays a dialog that allows the user to set up all the reminder information: date, time, message, alarm status, etc. The reminder statement has two parameters: the field containing the reminder, and the field containing the message.



Any time the user wants to edit a reminder the reminder statement should be used. You may want to set up a procedure that gets triggered whenever the user double clicks on the reminder display. This procedure only needs a single line:

```
reminder Reminder,Message
```

The reminder dialog has a subdialog for setting up repeating reminders.



Using this subdialog you can set up reminders that repeat every day, every week, every month, every quarter or once per year.

Reminder Functions

The functions listed below can build and work with reminders.

Function	Reference Page	Description																
reminder(date,time,type)	Page 5645	<p>This function builds a new reminder. There are three parameters: date, time and type. Date is the date for the new reminder (the function cannot create recurring reminders). However, you should not use a number here as you do with most date functions. You should use text that describes the date, for example "5/25/03" or "next tuesday". Time is the time for the new reminder. However, you should not use a number here as you do with most time functions. You should use text that describes the time, for example "5:22 pm". Type is the type of the new reminder: "a" for appointments or "t" for to-do's.</p> <p>This example adds a new reminder next tuesday at 2pm. The procedure stores this reminder in a field called Schedule:</p> <pre>addrecord Schedule=reminder("next tue","2:00 pm","a")</pre> <p>Another way to build a reminder is with the buildreminder statement, see reference page 5078. You can also edit a reminder with a user friendly dialog using the reminder statement (reference page 5645). This dialog allows you to set up recurring reminders (every tuesday, 15th of each month, etc.) and to configure alarms.</p>																
remindercaption(reminder)	Page 5649	<p>This function extracts the date from a reminder as formatted text that describes when the reminder will occur. The table below shows typical examples of how different reminder frequencies will be formatted by this function:</p> <table> <tbody> <tr> <td>Once only reminder:</td> <td>Tuesday, May 16th, 2004</td> </tr> <tr> <td>Annual reminder:</td> <td>August 8th of each year</td> </tr> <tr> <td>Quarterly reminder:</td> <td>First day of each quarter</td> </tr> <tr> <td>Monthly reminder:</td> <td>5th day of each month</td> </tr> <tr> <td></td> <td>Last day of each month</td> </tr> <tr> <td></td> <td>2nd Wed of each month</td> </tr> <tr> <td>Weekly reminder:</td> <td>Tue of each week</td> </tr> <tr> <td>Daily reminder:</td> <td>Every day</td> </tr> </tbody> </table> <p>The formula below displays the reminder date in a field called Schedule. This formula could be used in an auto-wrap text object or a Text Display SuperObject™.</p> <pre>remindercaption(Schedule)</pre>	Once only reminder:	Tuesday, May 16th, 2004	Annual reminder:	August 8th of each year	Quarterly reminder:	First day of each quarter	Monthly reminder:	5th day of each month		Last day of each month		2nd Wed of each month	Weekly reminder:	Tue of each week	Daily reminder:	Every day
Once only reminder:	Tuesday, May 16th, 2004																	
Annual reminder:	August 8th of each year																	
Quarterly reminder:	First day of each quarter																	
Monthly reminder:	5th day of each month																	
	Last day of each month																	
	2nd Wed of each month																	
Weekly reminder:	Tue of each week																	
Daily reminder:	Every day																	
remindercompare(reminder,date)	Page 5650	<p>This function checks to see if a reminder occurs on a specified date. Reminder is the reminder you want to compare. Date is the date you want to compare with the reminder. This function returns either true or false. It will return true if the reminder will occur on the specified date, including a repeating reminder that falls on that date.</p> <p>The example procedure below uses this function to select all reminders that will occur on next tuesday.</p> <pre>select remindercompare(Schedule,date("next tue"))</pre>																

Function	Reference Page	Description
reminderdate(reminder)	Page 5651	<p>This function extracts the date from a reminder. If a reminder repeats, the function will try to come up with the most appropriate single date. The table below shows how different reminder frequencies will be handled by this function:</p> <pre> Once only reminder: Actual date Annual reminder: Next occurrence of this reminder Quarterly reminder: 0 Monthly reminder: 0 Weekly reminder: 0 Daily reminder: 0 </pre>
reminderpriority(reminder)	Page 5652	This function extracts the priority of a to-do reminder (completed/not completed) from a reminder. The function returns a number from 0 to 3: 0 for the lowest priority to 3 for the highest priority.
remindertime(reminder)	Page 5653	This function extracts the time (number of seconds since midnight) from a reminder. The result of this function can be used with functions like timepattern((see “ Converting Between Times and Text ” on page 113).
remindertodo(reminder)	Page 5654	This function extracts the status of a to-do reminder (completed/not completed) from a reminder. The function returns a number: 0 if the to-do has not been completed (or the reminder is an appointment type), or 1 if the to-do has been completed.
remindertype(reminder)	Page 5655	This function extracts the type (to-do or appointment) from a reminder. This function returns a number: 0 if the reminder is an appointment type, or 1 if it is a to-do.

Alarms

If you have the optional Team Alarm extension installed (Mac OS 9 only), you can be notified of your reminders even when Panorama is not currently running. To do this, the Team Alarm extension keeps a separate private list of pending alarms. This list is in a special format that cannot be accessed by Panorama. However, this extra alarms database is updated automatically when a reminder is updated with the **reminder** statement (see “[REMINDER](#)” on page 5645 of the *Panorama Reference*). However, if you modify a reminder yourself without using the **reminder** statement, you’ll need to make sure the Team Alarm list is also updated. There are three statements for doing this: **alarmedit** (see “[ALARMEDIT](#)” on page 5023 of the *Panorama Reference*), **alarmdelete** (see “[ALARMDELETE](#)” on page 5022 of the *Panorama Reference*), and **alarmreset** (see “[ALARMRESET](#)” on page 5024 of the *Panorama Reference*).

True/False Formulas

In Panorama as in most programming languages, control flow decisions are made on the basis of formulas that are either true or false. The most basic true/false formula compares two values to see if they are equal.

```
PaymentMethod="C.O.D."
```

This formula will compare the value in the field `PaymentMethod` with `C.O.D.` The result will be true if `PaymentMethod` is `C.O.D.`, and false if it contains anything else (for example `Check`, `Cash`, `Visa`, etc.).

Comparison Operators

Panorama has about a dozen different operators that can compare two values and produce a true false result. You can type these operators in yourself (see “[Special Characters](#)” on page 57), or you can use the **Operator** sub-menu in the Function menu to type in the symbols for you. The table below lists the universal comparison operators. These comparison operators will work with any type of data: text, numeric, or date.

Operator	Example	True/False Meaning	Notes
=	A=B	is A equal to B?	
≠	A≠B	is A not equal to B?	Not available on PC
<>	A<>B	is A not equal to B?	
>	A>B	is A greater than B?	
≥	A≥B	is A greater than or equal to B?	Not available on PC
>=	A>=B	is A greater than or equal to B?	
<	A<B	is A less than B?	
≤	A≤B	is A less than or equal to B?	Not available on PC
<=	A<=B	is A less than or equal to B?	

All of the above operators require that A and B be the same data type. In other words, you cannot directly compare numbers to text, or text to dates. If A and B are different types you must convert them to the same type before comparing them, using the `str()`, `val()`, `pattern()`, `date()` or `datepattern()` functions. See “[Converting Between Numbers and Strings](#)” on page 84 and “[Converting Between Dates and Text](#)” on page 107 for more information on these functions.

Panorama also has a number of specialized comparison operators that work only with the text data type.

Operator	Example	True/False Meaning
<code>beginswith</code>	A beginswith B	does A begins with B?
<code>endswith</code>	A endswith B	does A end with B?
<code>contains</code>	A contains B	does A contain B?
<code>notcontains</code>	A notcontains B	does A not contain B?
<code>soundlike</code>	A soundlike B	does A sound like B (phonetically)?
<code>match</code>	A match B	does A match the wildcard pattern in B (disregarding upper/lower case)?
<code>matchexact</code>	A matchexact B	does A exactly match the wildcard pattern in B?
<code>notmatch</code>	A notmatch B	does A not match the wildcard pattern in B (disregarding upper/lower case)?
<code>notmatchexact</code>	A notmatch B	does A not exactly match the wildcard pattern in B?

Each of these operators deserves a more complete explanation, so here they are.

A beginswith B

This operator checks to see if the text in A begins with the characters in B. For example, the formula below will determine if the **Name** begins with the letters **Dr.** .

```
Name beginswith "Dr."
```

This formula will be true if the name is **Dr. Robert Johnson**, and false if the name is **Mark Reynolds**.

Note: The beginswith operator does not worry about upper or lower case, so **DR. ROBERT JOHNSON** or **dr. robert johnson** will also produce true results. If upper and lower case are important to you use the **matchexact** operator.

A endswith B

This operator checks to see if the text in A ends with the characters in B. For example, the formula below will determine if the **Name** ends with the letters **D.D.S.** .

```
Name endswith "D.D.S."
```

This formula will be true if the name is **Ronald Nelson, D.D.S**, and false if the name is **Mark Reynolds**.

Note: The endswith operator does not worry about upper or lower case, so **ronald nelson, d.d.s.** would also produce a true result. If upper and lower case are important to you use the **matchexact** operator.

A contains B

This operator checks to see if the text in A contains the characters in B. For example, the formula below will determine if the **Address** contains the letters **box**.

```
Address contains "box"
```

This formula will be true if the address is **P.O. Box 5328**, and false if the address is **6389 E. Wilson Blvd**.

Note: The contains operator does not worry about upper or lower case, so **P.O. BOX 5328** and **p.o. box 5328** would also produce true results. If upper and lower case are important to you use the **matchexact** operator.

A notcontains B

This operator checks to see if the text in A does not contain the characters in B. This is the exact opposite of the contains operator. For example, the formula below will determine if the **Address** contains the letters **box**.

```
Address notcontains "box"
```

This formula will be true if the address is **6389 E. Wilson Blvd**, and false if the address is **P.O. Box 5328**.

Note: This same function could also be performed by combining the not operator with the contains operator in the formula: **not (Address contains "box")**.

A soundslike B

This operator checks to see if the text in A “sounds like” the text in B. For example, the formula below will determine if the **LastName** sounds like the name **Smith**.

```
LastName soundslike "Smith"
```

This formula will be true if the name is **Smith, Smyth** or **Smythe**, and false if the name is **Jones** or **Williams**.

The method Panorama uses to determine whether two values sound alike is called “soundex.” This technique is not very exact, and often will produce extra matches that you might not think really sound similar. However, it almost never fails to match on names that do sound similar, so it is a good starting point when you are not sure of an exact spelling.

The soundex technique does require that the first letter of the two values match. For example even though we think they sound alike, **Christy** and **Kristy** will not match because the first letter is different.

A match B

This operator checks to see if the text in A matches a pattern you specify in B. The pattern allows you to set up very flexible “wildcard” matches where some characters must match and some don’t have to.

The pattern should combine normal characters, which must match the text in A, and wildcard characters: ? and *. The ? wildcard character will match any character. The * wildcard character (asterisk) will match a variable number of characters. The best way to understand wildcard matches is probably to look at a few examples.

Our first example uses the pattern `j*johnson`. With this pattern the name must begin with j (or J) and end with johnson (or Johnson, etc.) The characters in between don’t matter.

```
Name match "j*johnson"
```

This formula will produce a true result for names like `Jim Johnson`, `Jack Johnson`, `Joe Johnson`, etc. The formula will also be true for names like `J346 Ujohnson` or `J@#opcjohnson`.

The second example uses the pattern `926??`. With this pattern the zip code must begin with 926 and must be 5 digits long. (Our example assumes that `ZipCode` is a text field, not a numeric field.)

```
ZipCode match "926???"
```

This formula will produce a true result for zip codes like `92631` or `92685` but a false result for zip codes like `89324` or `92685-0301`. Here’s a variation that will work with 5 or 9 digit zip codes. The `??` characters mean that there must be at least five digits, while the `*` means that any extra characters are ok.

```
ZipCode match "926??*"
```

This formula will produce a true result for zip codes like `92631`, `92685` or `92685-0301`, but a false result for `926` or `9262`.

Don’t forget that a space is a normal character. The example below checks for people with a middle initial. The pattern looks for any number of characters followed by a space, followed by a single character, followed by a period, followed by another space, followed by any number of characters.

```
Name match "* ? . *"
```

This formula will produce a true result for `Robert E. Lee` or `Winston O. Link`, but a false result for `Frank Tesh`, `Billy Martin`, or `Sara Jessica Parkman`.

The match operator can be used to simulate the `beginswith`, `endswith` and `contains` operators. The table below shows the equivalent match formulas for each of these operators.

These formulas...	are the same as these.
<code>A match B+""</code>	<code>A beginswith B</code>
<code>A match ""+B</code>	<code>A endswith B</code>
<code>A match ""+B+""</code>	<code>A contains B</code>

Note: The match operator does not worry about upper or lower case. If upper and lower case are important to you, use the `matchexact` operator.

A matchexact B

This operator checks to see if the text in A matches a pattern you specify in B. This operator works exactly the same as the match operator, except that the normal characters must match exactly, including upper and lower case. For example, the formula below

```
Name matchexact "J*Johnson"
```

will produce a true result for **Jeff Johnson**, but a false result for **JEFF JOHNSON**. (However, **JEFF Johnson** would produce a true result.)

You can use the matchexact operator instead of beginswith, endswith, or contains if you need an exact upper and lower case match.

A notmatch B**A notmatchexact B**

These operators are the exact opposite of match and matchexact.

A like B

This operator checks to see if the text in A matches a pattern you specify in B. This operator is similar to the matchexact operator, but it uses different wildcard characters: % and _ instead of * and ? Here are some examples showing both formats:

```
Name matchexact "J*Johnson"
Name like "J%Johnson"

Zip matchexact "926???"
Zip like "926__%"
```

The like operator is included for compatibility with SQL servers. The like operator can be used for selecting a subset from a SQL master file, the match operator cannot.

Combining Comparisons

The basic comparisons described in the previous section can be combined together for more complicated decisions. There are four basic operators that can combine or modify decisions: **and**, **or**, **xor**, and **not**.

Operator	Sample	Description
and	A and B	true if both A and B are true
or	A or B	true if either A or B are true
xor	A xor B	true if A and B are different
not	not A	true if A is false

A and B

The **and** operator combines two true/false formulas together so that the result is only true if both formulas are true. The example procedure below determines if a person is a teenager.

```
if Age≥13 and Age<20
  Status="Teenager"
endif
```

The result of the formula is only true if the person is 13 or older and less than 20.

A or B

The **or** operator combines two true/false formulas together so that the result is true if either one of the two formulas are true. The example below determines if a transaction is being paid with a credit card.

```
if PaymentMethod="Visa" or PaymentMethod="MasterCard"
  Terms="Credit Card"
endif
```

The result of the formula is only true if the payment method is **Visa** or **MasterCard**.

Notice that each side of the **or** operator must contain a complete formula. The formula below looks right in English, but will not work in Panorama. The example below is **WRONG**:

```
if PaymentMethod="Visa" or "MasterCard"      /* WILL NOT WORK !! */
```

There must be a comparison on both sides of the **or**, as shown in the first example.

A xor B

The **xor** (short for exclusive-or) operator is a bit tricky. **Xor** combines two true/false formulas together so that the result is true if one of the two formulas is true, but false if both are true or both are false. Another way to put it is that the result will be true if A and B are different, but false if they are the same. The example below determines if two shoes are a pair.

```
if Shoe1="Left" xor Shoe2="Left"
  message "These shoes are a pair"
endif
```

The result of the formula is only true if one shoe is **Left** and the other shoe is **Right** (or to be more precise, not **Left**).

not A

The not operator reverses a true-false formula. If the result was true, now it will be false. If it was false, now it will be true.

```
if (not (Shoe1="Left" xor Shoe2="Left"))
  message "These shoes are not a pair!"
endif
```

Note: This example shows that if **not** is used as the very first operator in a formula in a procedure, you must surround the entire formula with an extra pair of parentheses. If **not** is in the middle of the formula the extra parentheses are not necessary. The parentheses are also not necessary if the formula is not in a procedure (in the Design Sheet or a **Formula Fill** dialog, for example).

Equals Comparison vs. Assignment

If you have skipped ahead to read about procedures you know that the equals sign is used to assign a value to a field or variable. The example formula we used earlier to compare two values:

```
PaymentMethod="C.O.D."
```

would also be the same formula used to assign the value C.O.D. to the field or variable PaymentMethod. At first glance this may appear ambiguous...the same formula is used to compare two values and to assign a value. How do we know when we are assigning and when we are comparing? The answer lies in the context in which the formula is found.

In a procedure, an assignment is always by itself, not part of a larger statement. A true-false formula is always part of another statement, for example **if**, **case**, **until**, **while**, **stoploopif**, **repeatloopif**, **find**, **select**. Here's an example that shows two formulas that look almost the same, but one is a true-false formula and one is an assignment.

```
if PaymentMethod="C.O.D."
    ShippingMethod="UPS"
endif
```

The first formula, **PaymentMethod="C.O.D."**, is part of the **if** statement. This formula means: Is the field (or variable) **PaymentMethod** equal to **C.O.D.** (true/false)?

The second formula, **ShippingMethod="UPS"**, is not part of any statement, but stands alone, so this is an assignment. The statement means: Take the value **UPS** and copy it into the field or variable named **Shipping-Method**.

If an assignment has more than one equals sign, the first equals sign is for the assignment and the rest are for comparisons. The example assignment below compares **B** and **C**. If they are equal (true) the value -1 will be copied into **A**. If they are not equal (false) the value 0 will be copied into **A**.

```
A=B=C
```

In other words, **A** becomes the result of the comparison between **B=C**, or **A = (B=C)**.

True/False Values

For purposes of calculation, Panorama treats true and false as numbers: true is -1 and false is zero. Panorama also has two functions that directly generate these values.

Function	Reference Page	Description
true()		This function always returns true (-1).
false()		This function always returns false (0).

Like any other number, you can store a true/false value in a field or variable and then use it later. The example below calculates whether a person is a teenager, then uses that information later.

```
local Teenager
Teenager=Age≥13 and Age<20
...
if Teenager
    Price=4.50
else
    Price=6.00
endif
```

Notice that the **if** statement doesn't need to compare, it simply uses the result of the comparison that was calculated earlier. In fact, the **if** statement (and all other statements that use true/false logic) can use any formula that produces a numeric integer result. The value 0 will be regarded as false, and any non-zero value will be regarded as true. The example below will be true if the length of the name is non-zero.

```
if length(Name)
    yesno "Is this a home address?"
    ...
endif
```

The first line of this example could also have been written **if length(Name)≠0**. The result is the same either way.

The ? Function

The ?(function allows a formula to make a decision. Will it be door number 1 or door number 2? The function uses a true-false value to pick from one of two values. The syntax for this function is like this.

```
?(decision-value,true-value,false-value)
```

The first parameter, **decision-value**, is used to pick which of the two choices will be returned as the final value, the **true-value** or the **false-value**.

For example, the formula below can be used to calculate a 10% discount if the quantity is 100 or more—

```
?( Qty<100 , Price , Price*0.9 )
```

The decision is based on the comparison **Qty<100**. If Qty is less than 100, the ? function picks the second parameter, **Price**. But if the quantity is 100 or more, the ? function will pick the third parameter, **Price*0.9**, for a 10% discount.

If you need to pick from three or more choices you can nest several ? functions together. For example, this formula shows how you can add a third discount level (20% for quantities of 500 or more)—

```
?( Qty<100 , Price , ?(Qty<500 , Price*0.9 , Price*0.8 ) )
```

Although these examples have used numeric data, text can also be used for either the true-false logic or the choices. The formula below, for example, could be used by a movie theater to check if a person is a child or an adult.

```
?( Age≤12 , "Child" , "Adult" )
```

Note: The ? function always evaluates all three parameters you give it, even though it really uses only two of the parameters. This means that you cannot use the ? function to avoid errors (for example divide by zero errors) because the error will happen before the ? function decides which parameter to use (use the `divzero()` function to avoid divide by zero problems).

Converting a Boolean Value to Text

The `boolstr()` function converts a boolean value to text, either **true** or **false**. For example

```
message boolstr(Qty<100)
```

will display **true** if the Qty is less than 100, or **false** if it is greater or equal to 100.

Linking With Another Database

Many database applications require multiple database files working together. For example, organizing a company's order entry operations usually requires an invoice file, an inventory/price list file, and possibly a customer file. The primary method for accessing information in other databases is the **lookup()** function (and other related functions). This function can search for and retrieve information from any open database. Need to look up a price or a customer's credit limit? Chances are the lookup() function is the tool for the job.

When you look up information manually (for example, looking up someone's number in the phone book), you are actually performing a multi-step process. You start with one piece of information—a person's name, for example. The first step is to locate the correct phone book. Once you've located the correct book, you must search through it to find the name of the person you are looking for. When you find the name, the final step is to copy down the person's phone number.

Panorama's lookup() function follows a similar process when it looks up data. For example, suppose you want to find out the number of calories in an orange using the database shown here.

Groceries											
Fruit	Serving Size	Calories	Fat (Total Fat	Saturated	Cholesterol	Sodium	Potassium	Carbc	Dietary Fiber	
Grapes	1-1/2 cups grapes	90	10	1g	0g	0mg	0mg	270mg	24g	1g	
Lemon	1 med lemon (58g)	15	0	0g	0g	0mg	0mg	0mg	5g	1g	
Lime	1 medium (67g)	20	0	0g	0g	0mg	0mg	75mg	7g	2g	
Cantaloupe	1/4 melon (134g)	50	0	0g	0g	0mg	25mg	280mg	12g	1g	
Honeydew	1/10 melon	50	0	0g	0g	0mg	35mg	310mg	13g	1g	
Orange	1 med orange (154g)	70	0	0g	0g	0mg	0mg	260mg	21g	7g	
Tomato	1 med tomato (148g)	35	0	1g	0g	0mg	5mg	360mg	7g	1g	
Pear	1 medium (166g)	100	10	1g	0g	0mg	0mg	210mg	25g	4g	
Kiwifruit	2 med kiwi (148g)	100	10	1g	0g	0mg	0mg	0mg	24g	4g	
Grapefruit	1/2 grapefruit	60	0	0g	0g	0mg	0mg	230mg	16g	6g	

Here is the formula for looking up the number of calories in an orange. The parameters to the lookup contain all the information necessary to locate the information.

lookup database → `lookup("Groceries",Fruit,"Orange",Calories,0,0)`

lookup data field → `Calories`

lookup key field → `Fruit`

lookup key value → `"Orange"`

lookup default value → `0,0`

lookup summary level (almost always zero) → `0,0`

The first parameter is called the **lookup database**. It tells Panorama what database to look in for the information, in this case **Groceries**.

The second and third database tell Panorama how to search for the data you want. In this case Panorama is being told to "search through the Fruit column until you find **Orange**." The field to look in (in this case **Fruit**) is called the **lookup key field**. The data to look for (in this case **Orange**) is called the **lookup data value**. By the way, Panorama is very picky about the lookup data value. It must exactly match the value in the database, or Panorama won't find a match. In this case only Orange will work — not **orange** or **ORANGE** or even **oRaNGe**!

At this point we come to a fork in the road. Perhaps Panorama found **Orange** in the database, perhaps not. If it did the fourth parameter tells Panorama what to do next. This fourth parameter is called the **lookup data field**, and it may be any field in the lookup database. In this case it is **Calories**, so Panorama will lookup the value in the **Calories** field (**70**) and return it as the result of the function.

What if Panorama didn't find **Orange** in the database? In that case Panorama simply returns the value of the fifth parameter, the **lookup default value**. In this case the default value is **0**. The default value should match the data type of the lookup data field. Since **Calories** is a numeric field, the default is also numeric. If the lookup data field had been a text field (for instance **Serving Size**) the default would need to be text (for example "").

The sixth and final parameter to the lookup function is the **lookup summary level**. This is the minimum summary level to be searched within the lookup database. Usually the lookup summary level is zero so that the entire lookup database will be searched. If the level is set to 1 through 7, only summary records will be searched. This is useful if you want to look up summary information (see "[3-Step Summarizing](#)" on page 365 of the *Panorama Reference*) while ignoring the raw data.

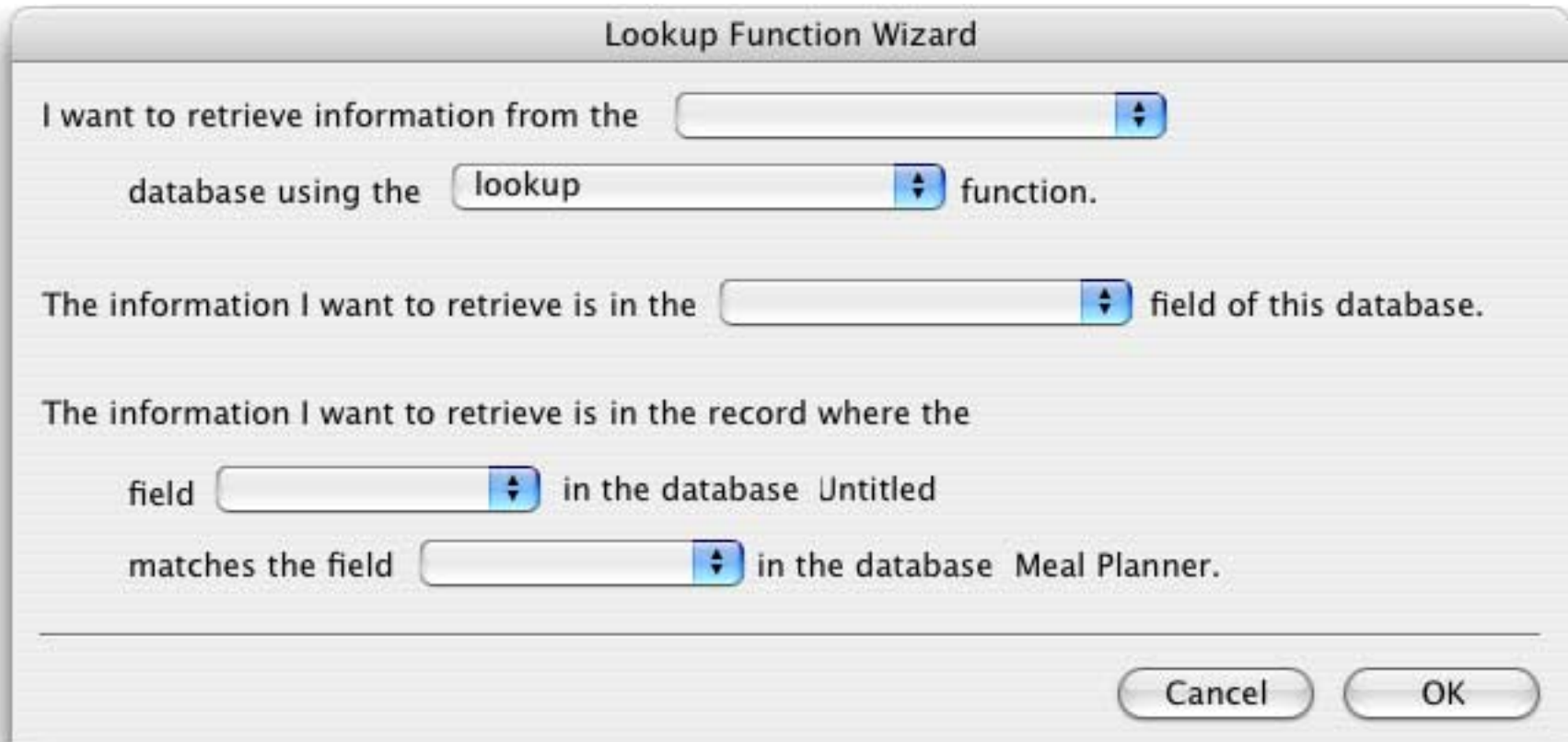
In this example the end result of the lookup is the value **70**. The lookup() function is often used by itself, but a more complicated formula can take this value and perform additional computations. If the result of the lookup is a text value then all of the text functions described earlier in this chapter can be used to modify the result.

The Lookup Wizard

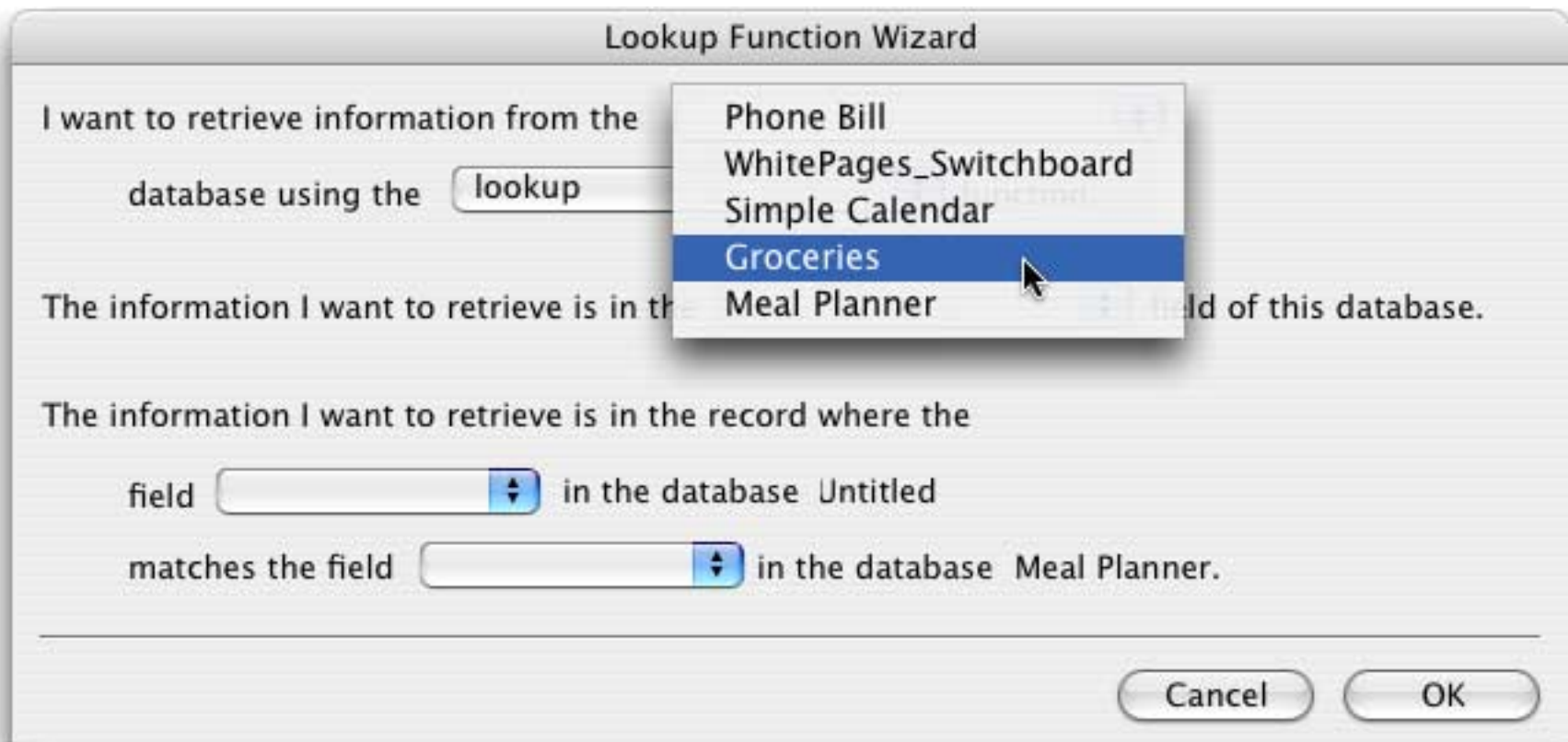
Since the lookup() function is kind of picky about all of its parameters we've provided a "fill-in-the-blanks" dialog to help build the function. To open this dialog simply pull down the **Functions** Menu and choose **lookup(...)**.



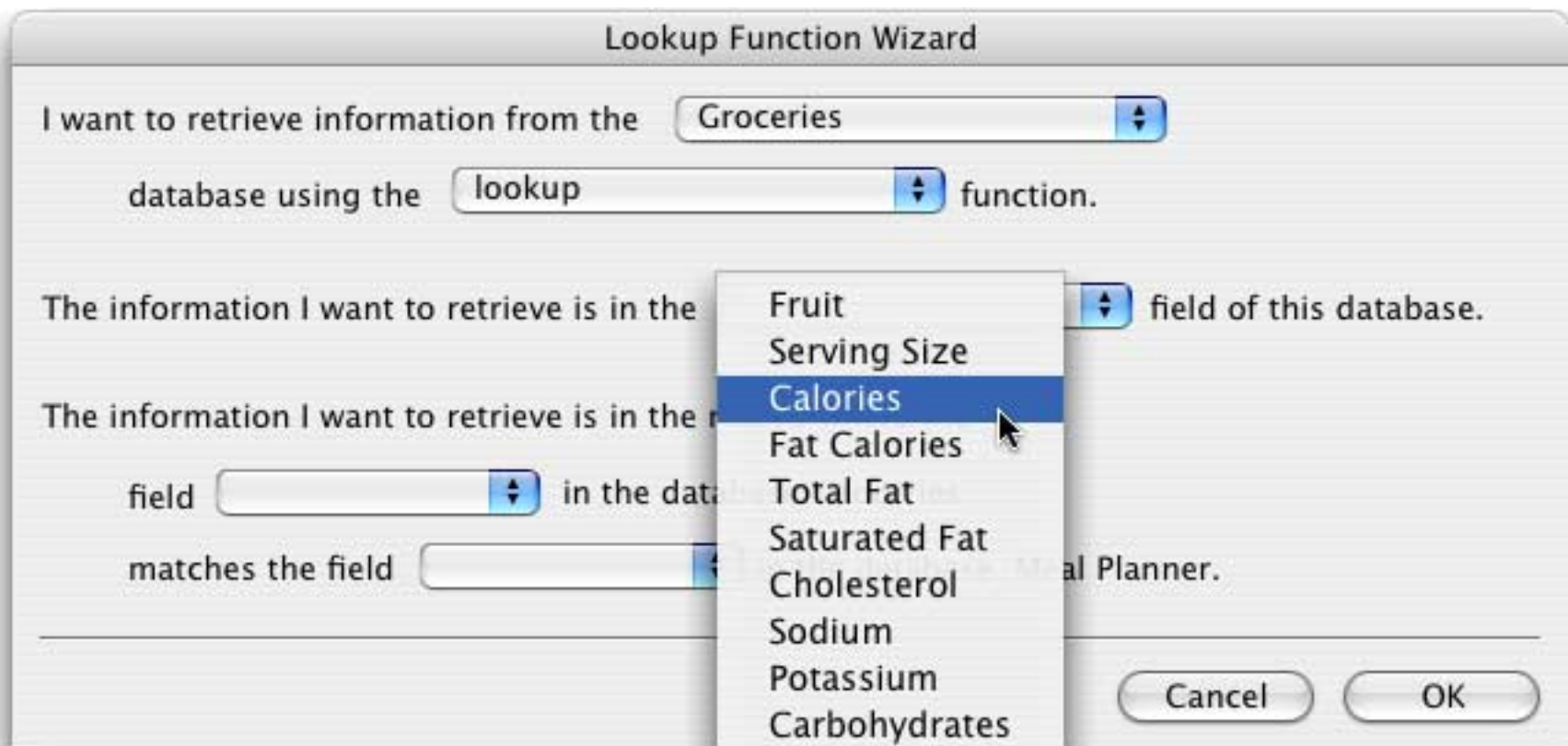
Now the lookup wizard dialog appears.



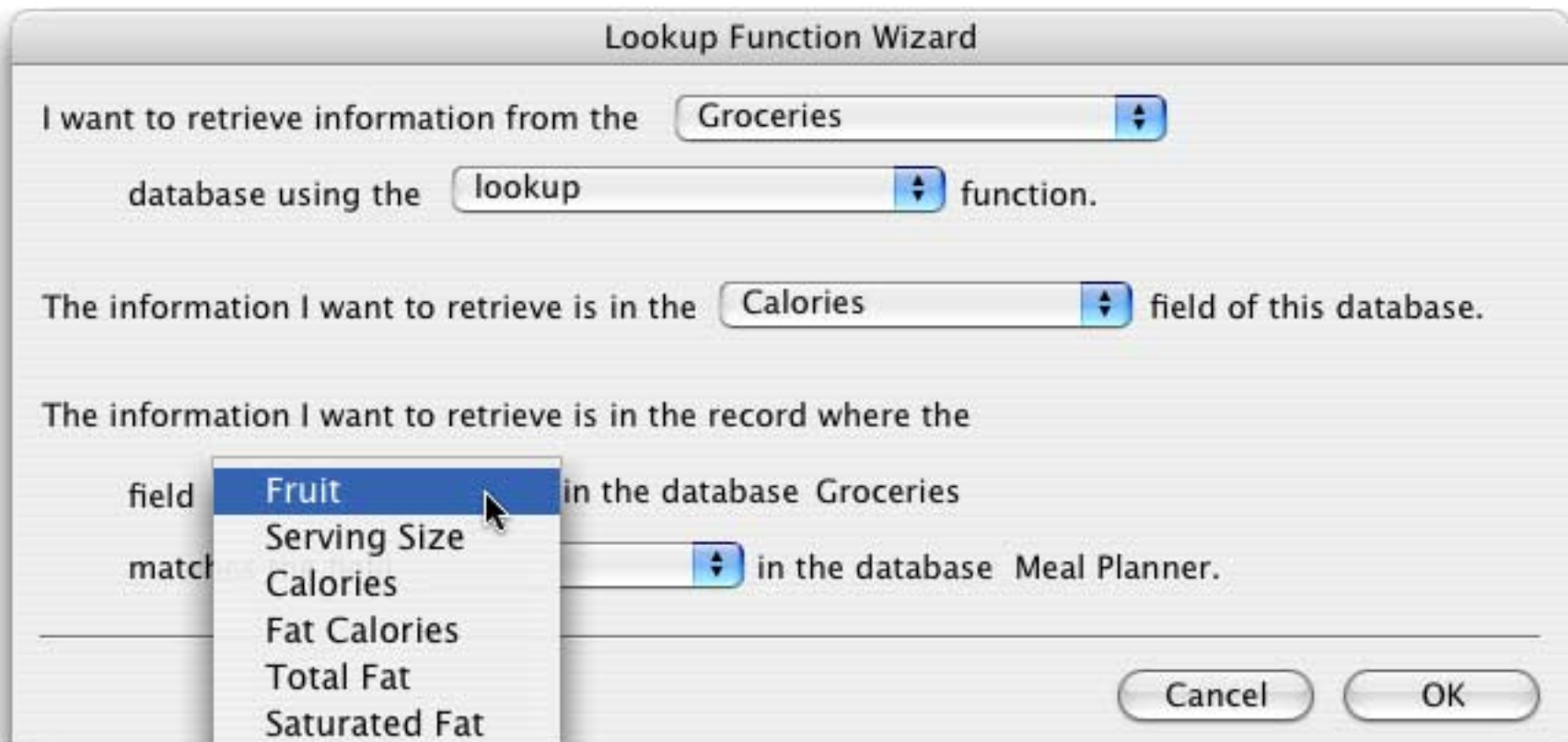
To create the lookup function, start at the top of the dialog and work your way down. Start by selecting the database to lookup from (in this case [Groceries](#)).



Next, choose the data you want to retrieve (the lookup data field, which will become the fourth parameter to the lookup function). In this case we want to retrieve the number of [Calories](#).



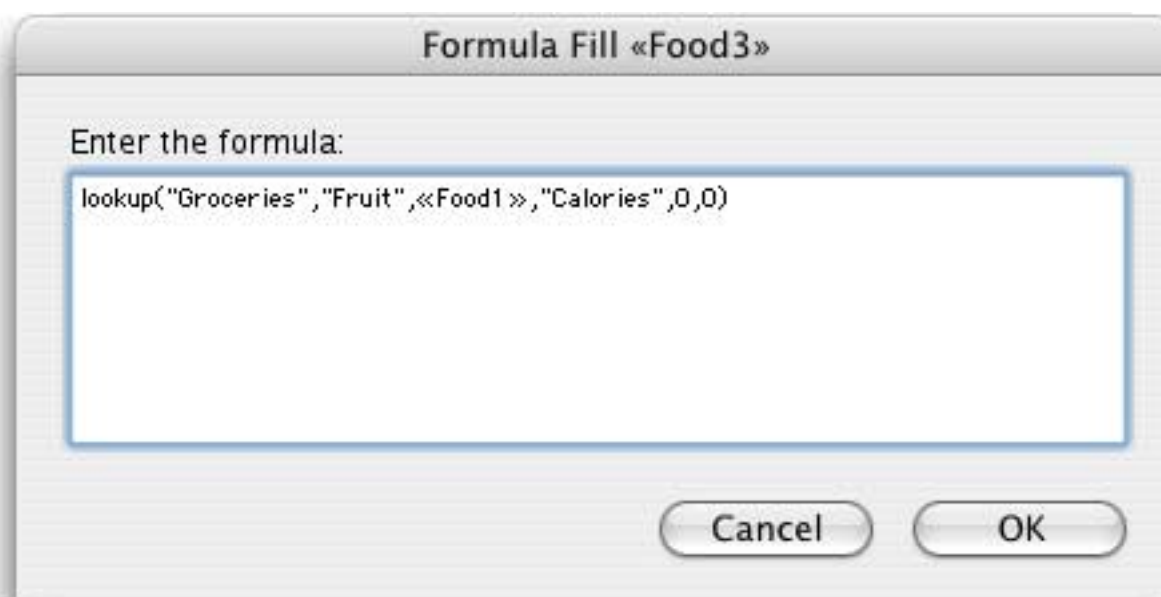
The next step is to choose the lookup key field, which in this case is [Fruit](#).



Finally, choose the field containing the lookup key value. If there is no such field (perhaps the value is in a variable) then just choose any field and adjust the formula once the wizard is finished.



Once you've made all of the selections press **OK** to generate the finished formula. For example, if you were using the Formula Fill command the formula would look like this:



Type Mismatch Problems

One of the most common problems when setting up a lookup function is type mismatches. With some careful thought, however, you can avoid these problems.

The first source of type mismatch problems is the lookup key field and the lookup key value. The field and value must be the same type of data. In other words, if the lookup key field is numeric, the lookup key value must be numeric also. If necessary, you can convert a text key value into numeric with the `val()` function, or you can convert a numeric key value into text with the `str()` function (see [“Converting Between Numbers and Strings”](#) on page 84 for details on both of these functions).

```
lookup("Catalog","Part#",val(Item),"Price",0,0)
```

Another source of type mismatch problems is the lookup data field. This field must have the same type of data as the field you want to store the result in. For example if you look up a price, the result must be stored in a numeric field.

If you need to store a numeric value in a text field, use the `str()` function to convert the value. The `str()` function should go outside the entire lookup function, for example

```
str(lookup("Catalog","Item",Desc,"Price",0,0))
```

Another source for type mismatch problems is the lookup default value. The default value should be the same type as the lookup data field. If the lookup data field is numeric, the default should be numeric (for example 0 or 100). If the lookup data field is text, the default should also be text (for example "" or "n/a").

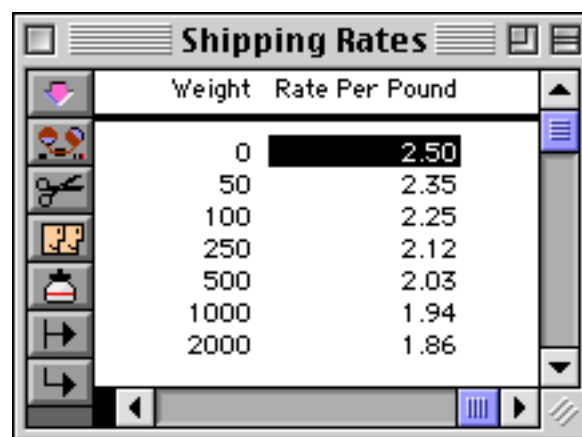
Lookup Variations

There are actually several different variations of the lookup function. All of the variations have the same six parameters. The standard lookup function locates the first occurrence of the key (nearest to the beginning of the file).

Function	Reference Page	Description
lookup(Page 5500	This function searches for the first occurrence of the value within the lookup database. If there is more than one copy of the value in the database this function will find the one closest to the top.
lookuplast(Page 5505	This function searches for the last occurrence of the value within the lookup database. If there is more than one copy of the value in the database this function will find the one closest to the bottom. However, there is one exception. If you are looking up within the current database Panorama will skip the current record. If the current record matches the key value then Panorama will skip backwards to the next matching record.
lookupselected(Page 5513	This function searches for the first occurrence of the value within the selected records in the lookup database. Unselected (invisible) records are ignored. If there is more than one copy of the value within the selected records this function will find the one closest to the top.
lookuplastselected(Page 5507	This function searches for the last occurrence of the value within the selected records in the lookup database. Unselected (invisible) records are ignored. If there is more than one copy of the value within the selected records this function will find the one closest to the bottom.
table(Page 5827	The table function allows you to lookup data by an approximate match instead of exact match. If the table function does not find an exact match, it uses the next lower value. A common example is a shipping rate table. Rate tables do not have an entry for every possible weight. Instead, the table only lists weights where the shipping rate changes. For example, suppose a rate table contains entries for 100 pounds and 250 pounds, and you have a 158 pound package. The table function will return the rate for the next lower value, in this case the 100 pound rate.

Looking Up Rates in a Rate Table

The table() function is designed for looking up rates from a table. For example, this function can be used to look up shipping rates, tax rates, discount rates, or any kind of stepped rate where the rate changes according to a sliding scale. To illustrate this function, consider this shipping rate database.



Weight	Rate Per Pound
0	2.50
50	2.35
100	2.25
250	2.12
500	2.03
1000	1.94
2000	1.86

For packages from 0 to 49.99 pounds the rate is 2.50 per pound. For packages from 50 to 99.99 pounds the rate is 2.35 per pound, from 100 to 249.99 the rate is 2.25 etc. Suppose we use a regular lookup function to look up the weight, like this.

```
lookup("Shipping Rates",Weight,PackageWeight,«Rate Per Pound»,0,0)
```

This formula will work fine for weights that appear in the table like 50, 100 and 250. But for other weights like 47 or 182 the formula will return the default value, zero. To fix this, use the table function instead of the lookup function.

```
table("Shipping Rates",Weight,PackageWeight,«Rate Per Pound»,0,0)
```

The table function will return the closest lower match. This means that if the **PackageWeight** is 3, 17 or 42 the formula will return 2.50. If the **PackageWeight** is 110 or 246 the formula will return 2.25, etc. Here is a complete formula that calculates the shipping cost for any package.

```
PackageWeight*table("Shipping Rates",Weight,PackageWeight,«Rate Per Pound»,0,0)
```

The formula looks up the rate per pound and then multiplies that rate by the package weight.

Looking Up Multiple Fields From One Record

Sometimes you may need to lookup several fields in the same record. For example, when you lookup someone's address you may also want to lookup their city, state, zip code, phone number and recent purchasing history. In a procedure one way to do this is with multiple **lookup()** functions, like this.

```
Address=lookup("Customers",Company,Company,Address,"",0)
City=lookup("Customers",Company,Company,City,"",0)
State=lookup("Customers",Company,Company,State,"",0)
Zip=lookup("Customers",Company,Company,ZipCode,"",0)
Phone=lookup("Customers",Company,Company,"Phone#","",0)
```

When a procedure contains several **lookup()** in a row for the same thing like this Panorama doesn't actually search the database over and over again. Instead it notices that it is searching for the same item and simply grabs the data from the record it has already found.

To make multiple field lookups even faster you can use the `speedcopy` statement (see “[SPEEDCOPY](#)” on page 5784 of the *Panorama Handbook*). (Remember, since this is a statement it can only be used within a procedure, see “[Procedures](#)” on page 203). The `speedcopy` statement can transfer many fields at once, but only if the fields to be copied in the two databases match exactly. The fields to be copied must appear in exactly the same order in both databases, and the fields must have the same data types. With all these restrictions, you may be surprised to find out that the fields do not have to have the same names!

Here’s how `speedcopy` works. Before you use `speedcopy`, you must perform an assignment with a `lookup(` function (or a variation of the `lookup(` function: `lookuplast(`, `lookupselected(`, etc., see “[Lookup Variations](#)” on page 136). The `lookup(` function locates the record containing the information to be copied. Once the record has been located the `speedcopy` statement can be used to copy the additional data.

The `speedcopy` statement has three parameters.

```
speedcopy FirstAssignField,LastAssignField,FirstTargetField
```

The first two parameters are fields in the current database. The last parameter is a field in the target database. All of these field names should be surrounded by quotes (for example “`Name`”, not `Name`). `Speedcopy` starts by converting these field names into field numbers. For example, if a field would be the third column in the data sheet, it is field #3.

Once `speedcopy` has converted the field names into numbers, it starts copying data. Suppose the `FirstAssignField` was field number 3, and the `FirstTargetField` was field number 8. `Speedcopy` will start by copying field #8 in the target database into field #3 in the current database. Then it will copy field #9 in the target database into field #4 in the current database. It will continue copying fields until it has copied something into the `LastAssignField`.

To show a specific example, suppose we have two databases, `Organizer` and `Customers`, with the fields listed below:

	Organizer	Customers
1	Name	Company
2	Title	Address
3	Company	City
4	Address	State
5	City	ZipCode
6	State	Phone#
7	Zip	Fax#
8	Phone	Cust#

The procedure below will quickly copy the `Address`, `City`, `State`, `Zip` and `Phone` fields from the `Customers` database to the `Organizer` database.

```
Address=lookup("Customers",Company,Company,Address,"",0)
if Address<>" "
    speedcopy "City","Phone","City"
endif
```

Let’s take a close look at how this procedure works. The first line attempts to lookup the `Address` from the `Customer` database. If this lookup fails, the procedure is finished. However, if the lookup succeeds the procedure continues with the `speedcopy` statement.

The first parameter of the `speedcopy` statement is `City`, which is field #5 in the current database (`Organizer`). The second parameter is `Phone`, which is field #8 in the current database. The final parameter is `City`, which is field #3 in the target database (`Customers`).

In this example **speedcopy** will copy 4 fields from **Customers** into **Organizer**, as shown by the blue arrows in this table. The green arrow represents the original **lookup()**.

	Organizer	Customers
1	Name	Company
2	Title	Address
3	Company	City
4	Address	State
5	City	ZipCode
6	State	Phone#
7	Zip	Fax#
8	Phone	Cust#

As **speedcopy** moves data from one database to another, it doesn't make any kind of checks on the data. If the fields aren't really in the same order, **speedcopy** will cheerfully copy them in the wrong order. Even worse, if you try to copy a numeric field into a text field or a text field into a numeric field, **speedcopy** will not object, but will speedily turn your current database into swiss cheese. The moral of the story is to use the **speedcopy** statement very carefully. Like any sharp instrument you want to make sure it is pointed in the right direction before you use it.

The GrabData Function

The **grabdata()** function ([reference page 5327](#)) grabs the contents of a field in the current record of any open database. You can grab data from the current database, or from another database. The function has two parameters — the name of the database to grab from and the name of the field within that database. For example here is the formula to look up the number of calories of the currently selected fruit.

```
grabdata("Groceries",Calories)
```

The value returned by this function will change depending on what record is active in the **Groceries** database.

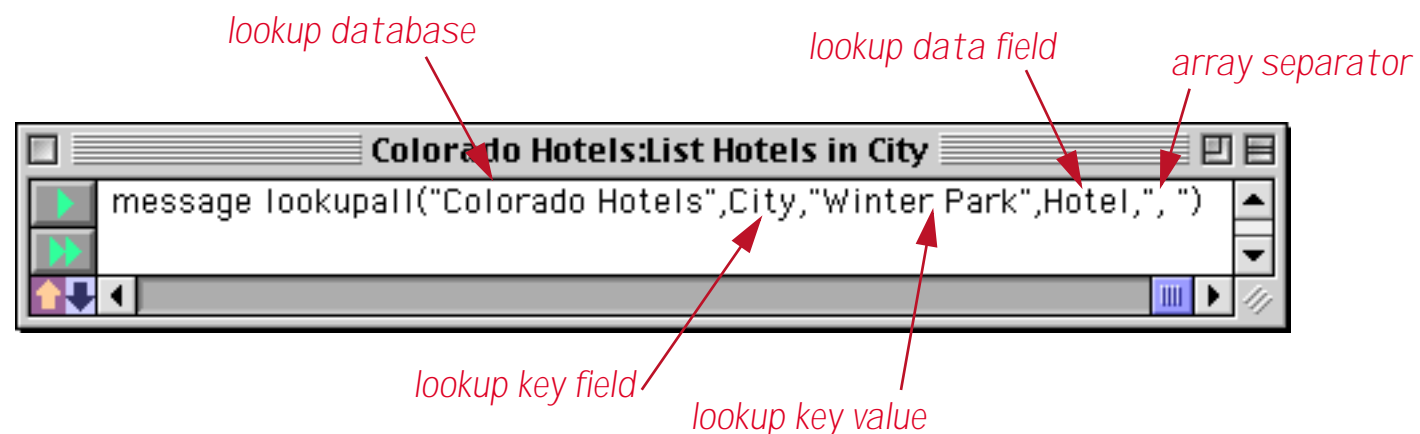
Looking Up Multiple Values at Once

The normal lookup functions return only a single value even if many records in the lookup database match the key value. The **lookupall()** function ([reference page 5502](#)) builds a text array containing one item for every record in the target database that matches. The function has five parameters. The first four parameters are the same as the other lookups with one slight difference: **lookup database**, **lookup key field**, **lookup key value** and **data field** (see "[Linking With Another Database](#)" on page 131). The difference is that the **lookup key field**

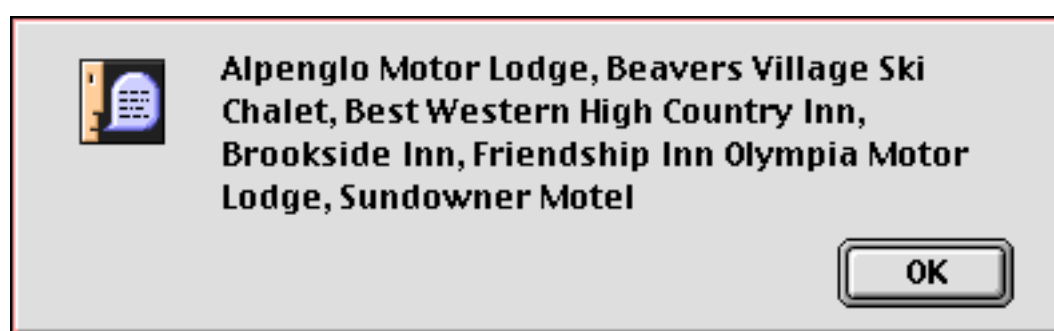
must be a text field — you cannot use a numeric or date field for the key. The fifth parameter is the separator character for the array that is constructed (see “[Text Arrays](#)” on page 93). Unlike most text array functions, the `lookupall()` function allows the separator to have more than one character. For example, you could use `,` to place a comma and space between each item. To illustrate this function, consider this database of hotels.

Hotel	City	Rate	Units	Phone	Stars
Apache Court	Colorado Springs	17.00	13	471-9440	1
Argo Motor Inn	Idaho Springs	36.00	17	567-4476	3
Aspen Lodge	Estes Park	74.00	23	586-4241	4
Aspen Motel	Loveland	24.00	16	667-0725	1
Aspen Square	Aspen	85.00	105	925-1000	4
B'n B Motel	Colorado Springs	19.00	17	598-3816	3
Beavers Village Ski Chal	Winter Park	128.00	148	726-5741	3
Bel Air Motel	Colorado Springs	20.00	18	598-7057	1
Bel Mar Motel	Pueblo	28.00	23	542-3268	3
Bel Rau Lodge	Cortez	26.00	1	565-3738	2
Bella Vista Court	Colorado Springs	20.00	13	633-4655	1

Using the `lookupall()` function we can create a procedure that lists all of the hotels in [Winter Park](#).



When you run this procedure it displays a list of all of the hotels in Winter Park, like this.

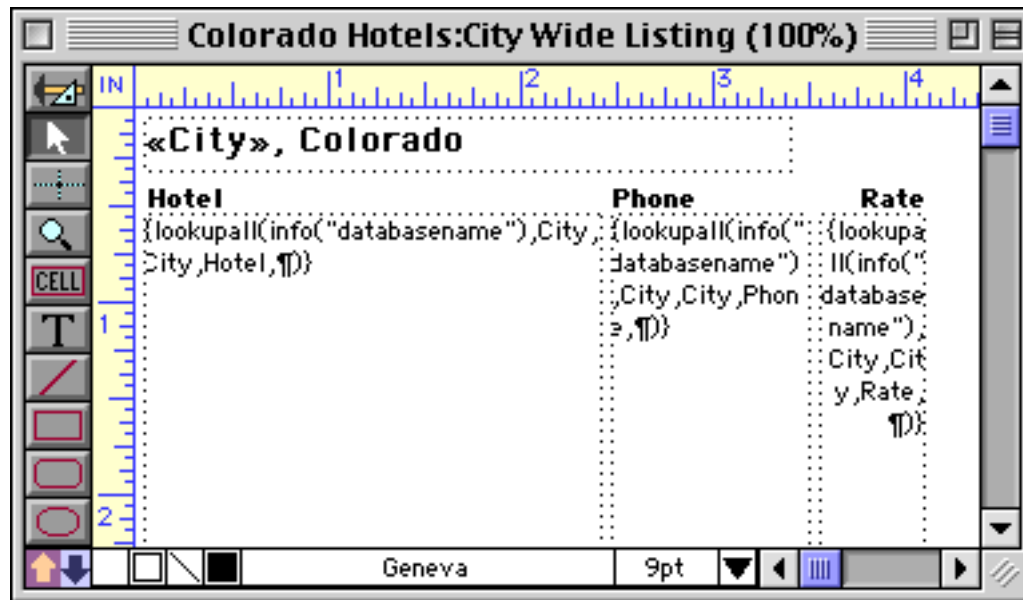


Of course you might want to list cities other than Winter Park. Here's a modified version of this formula that lists the hotels in whatever the current city is.

```
lookupall("Colorado Hotels",City,City,Hotel,", ")
```

Of course you could also display the list of hotels in an auto-wrap text object or Text Display SuperObject. The list would update automatically as you moved from record to record.

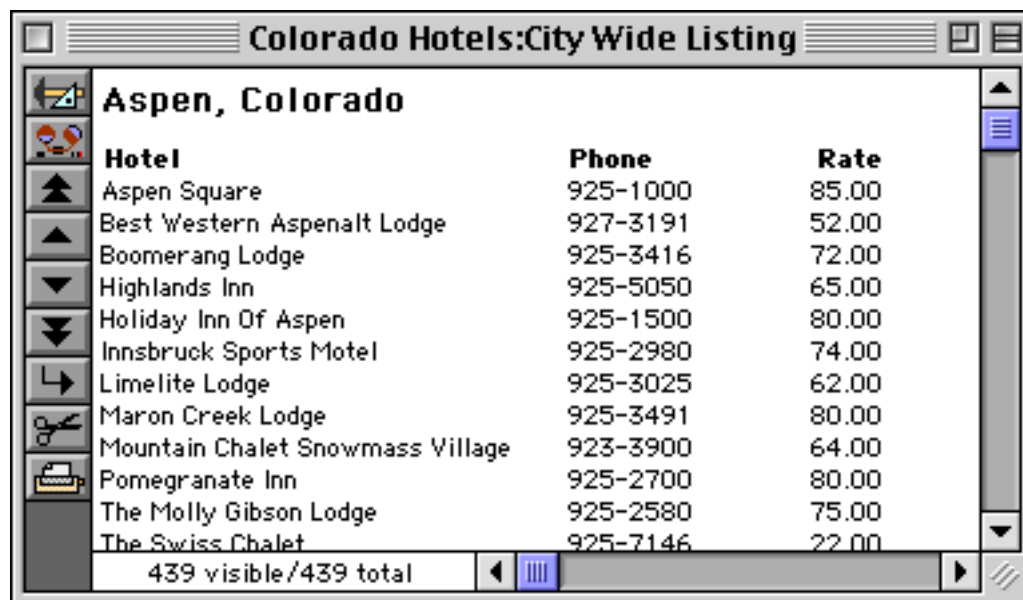
In our hotel example the lookup data field is a text field (**Hotel**). However, the **lookupall()** function will also work with numeric and date data fields. If the field is numeric or date it will be converted to text before it is added to the array. In this illustration three different **lookupall()** functions are used to display the hotel name, phone number, and rate.



When switched to Data Access mode you can see that all of the rates have been converted to text and appended together, one per line.



As you switch to a different record the form updates automatically.



In this case the lookup is from the current database, but any open database can be used.

Panorama also includes seven additional variations on the `lookupall()` function that lookup from two to eight fields from the target database, instead of just one. Each of these functions includes a parameter for controlling the separator between records and the separator between each fields.

Function	Reference Page	Description
<code>lookupalldouble(thedb, keyfield, keyvalue, datafield1, datafield2, mainsep, subsep)</code>		This function does a lookupall for two fields (double). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 and datafield2 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.
<code>lookupalltriple(thedb, keyfield, keyvalue, datafield1, datafield2, datafield3, mainsep, subsep)</code>		This function does a lookupall for three fields (triple). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 through datafield3 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.
<code>lookupallquadruple(thedb, keyfield, keyvalue, datafield1, datafield2, datafield3, datafield4, mainsep, subsep)</code>		This function does a lookupall for four fields (quadruple). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 through datafield4 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.
<code>lookupallquintuplet(thedb, keyfield, keyvalue, datafield1, datafield2, datafield3, datafield4, datafield5, mainsep, subsep)</code>		This function does a lookupall for five fields (quintuplet). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 through datafield5 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.
<code>lookupallsextet(thedb, keyfield, keyvalue, datafield1, datafield2, datafield3, datafield4, datafield5, datafield6, mainsep, subsep)</code>		This function does a lookupall for six fields (sextet). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 through datafield6 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.
<code>lookupallseptuplet(thedb, keyfield, keyvalue, datafield1, datafield2, datafield3, datafield4, datafield5, datafield6, datafield7, mainsep, subsep)</code>		This function does a lookupall for seven fields (septuplet). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 through datafield7 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.
<code>lookupalloctet(thedb, keyfield, keyvalue, datafield1, datafield2, datafield3, datafield4, datafield5, datafield6, datafield7, datafield8, mainsep, subsep)</code>		This function does a lookupall for eight fields (octet). The lookup is of all records in thedb where the keyfield matches the supplied keyvalue. For each record datafield1 through datafield8 are joined together with the subsep separator (which may be more than one character) and each record is joined together with the mainsep characters.

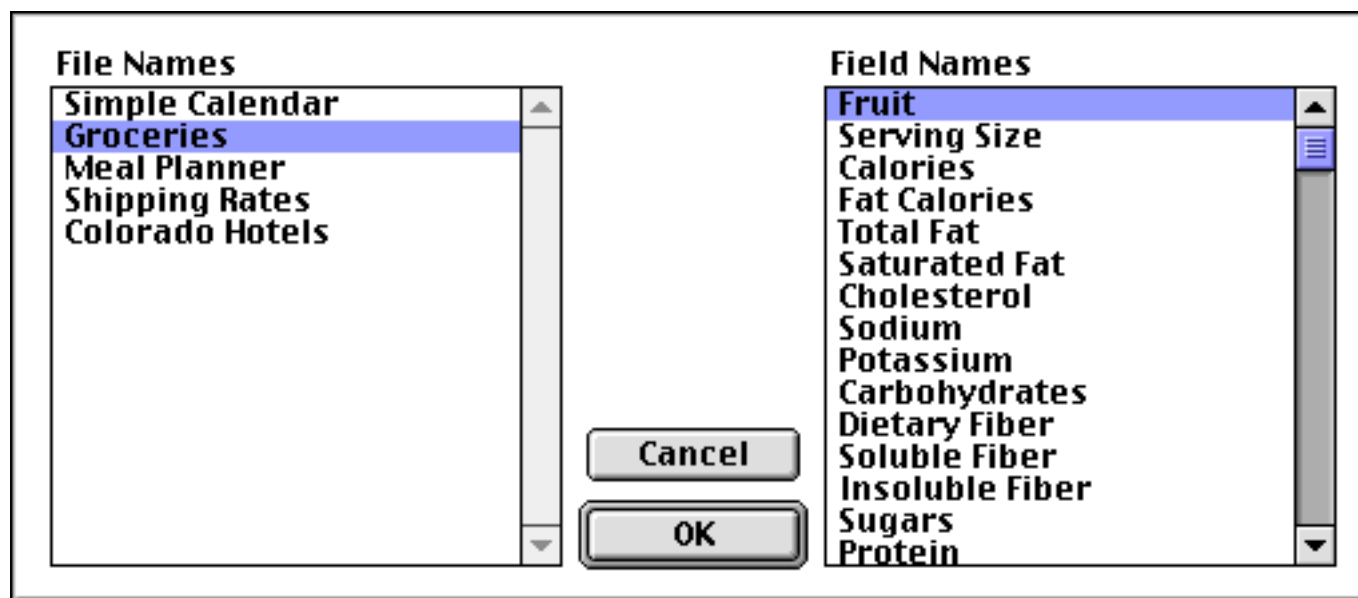
Linking Clairvoyance to the Lookup Key Field

Panorama's Clairvoyance feature anticipates what you are about to type by scanning the entries you have already made in the same database. When you are working with multiple files, you can configure Clairvoyance so that it scans the entries in another database instead (see "[Clairvoyance® Across Multiple Files](#)" on page 286 of the *Panorama Handbook*). This is called linking clairvoyance to another field.

There are two reasons for linking clairvoyance to another field. Clairvoyance cannot anticipate values until they have been typed in at least once. If all the possible values have already been entered into another database, Clairvoyance can start working immediately by looking into the other database.

Another advantage is speed. If your price list contains 200 records and your invoice database contains 2000 records, Clairvoyance can scan the price list 10 times faster. As your database gets larger, this speed difference may become noticeable.

To set up a clairvoyance link to another field, use the design sheet. Click on the name of the field you want to set up. Then choose **Set Up Link** from the Special Menu. Choose the database you want to link to, and the field within that database.



Press **OK** to enter the link into the design sheet. Like all other design sheet options, the link does not actually take effect until you tell Panorama to create a new generation. In the design sheet shown below five fields have been linked to the **Fruit** field in the **Groceries** database.

Field Name	Type	Dir	Align	Out	Inp	Range	Choi	Link	Clair	Tab	Cap	Dup	Def
Food1	Text	0	Left			Any		Groceries:Fruit	On	Off	Off	Yes	
Food2	Text	0	Left			Any		Groceries:Fruit	Off	Off	Off	Yes	
Food3	Text	0	Left			Any		Groceries:Fruit	Off	Off	Off	Yes	
Food4	Text	0	Left			Any		Groceries:Fruit	Off	Off	Off	Yes	
Food5	Text	0	Left			Any		Groceries:Fruit	Off	Off	Off	Yes	
Calories	Num	0	Right			Any			Off	Off	Off	Yes	

When you are editing data within a field that has a clairvoyance link set up, clairvoyance checks the characters you type against the data in the second database. When it finds a possible match, it enters the rest of the value for you.

Food1	Food2	Food3	Food4	Food5	Calories
Apple	Pear				180
Orange	Tomato				105
Kiwifruit	Orange	Grapefruit			230
Honeydew					50
Grapes	Lemon	Avocado			105

Looking Up Data in the Current File

You can use the `lookuplast()` function to look up the previous entry, with the same value, in the same database. For example, in a checkbook database you can automate repetitive payments by looking up the previous payment to the same company. By using the `info("database")` function to look up the database name you can make sure that the formula will continue to work even if the database is renamed.

```
lookuplast(info("database"),PayTo,PayTo,Amount,0,0)
```

Suppose that your last check to **Pacific Mutual** was **\$178.34**. Using the formula above you could automatically enter this value the next time you write a check to this company.

Another application for looking up data in the current file is locating summary information further down in the database. To do this, set the lookup summary level to a non-zero value so that only summary records will be located.

The Assign Function

The `assign()` function is so different from every other Panorama V function that we gave it its own section, all by itself. Instead of calculating a new value based on its parameters, the `assign()` function allows you to assign the result of a partial formula to a field or variable. This is called a "side effect" because this assignment is in addition to the normal operation of the formula that contains the `assign()` function (or multiple `assign()` functions).

The `assign()` function has two parameters: **value** and **destination**. The value parameter is simply any formula that calculates a value. The calculated value may be either numeric or a text value.

The destination parameter is the name of the field or variable to store the value into. If the destination is a field, the field must have the same data type as the calculated value (in other words, numeric values must be stored in numeric fields, text values in text fields. If the destination is a variable, the variable must already have been created with a local, global, fileglobal, permanent or windowglobal statement.

Let's look at some examples to illustrate the operation of this function. For example, consider the equation below, which might be part of a procedure. In this example, both A and B will be assigned the value X+Y.

```
B=assign(X+Y,"A")
```

When using the `assign()` function it's important to keep in mind that this function also produces a value - the value calculated by the value parameter. For example, in the example above B is also assigned the value X+Y. If you forget that the `assign()` function produces a value you may encounter unexpected error messages or even accidentally overwrite data.

The previous example illustrated the operation of the `assign()` function, but it wasn't really very useful. Here is a more interesting example. This example quickly scans a field and finds the largest value. The output of the `arraybuild` (the variable x), is simply ignored. The real result is the variable `biggest`, which is calculated as a side effect. At the end of the procedure this variable will contain the largest value in the current field.

```
local biggest,x
biggest=«»
arraybuild x,«,",assign(?(biggest>«»,biggest,«»),"biggest")
message biggest
```


The `assign()` function is very useful for formulas in forms, for example `TextDisplay` or auto-wrap text objects. In these objects the `assign()` function can be used for intermediate values that need to be used over and over. Of course when you use the `assign()` function this way you have to keep in mind the order of expression evaluation. You must make sure that the `assign()` function is evaluated before the destination is used. Here is an example where the customer name is assigned to a variable named `tempCustomerName` (which must be created with a global or fileglobal statement before this formula is displayed), then used in several locations in the text.

```
"Dear "+
assign(lookup("CustomerFile","CustomerNumber",custID,"Name","",0),"tempCustomerName")+","+"
"+tempCustomerName+", you are invited to particpate blah blah blah. "+tempCustomerName",
you won't want to miss out on this blah blah blah"
```

If you use the `?(` function in combination with the `assign()` function be careful because the `assign()` function will always be evaluated. It doesn't matter whether the `assign()` function is in the true or false part of the `?(` function.

When the `assign()` function is used with a field, the field display will not be updated automatically. You'll need to do that separately with `showfields`, `showcolumns`, etc. (depending on what operation you have done).

Note: Assigning a value to a field in a `formulafill` (or `select`) in a multi-user Partner/Server database doesn't work properly. The field on the local client will change, but the server will not be updated. There's no warning or error message, so you should avoid using this function with fields in a Partner/Server database.

Zip Code Lookup

If you have purchased Panorama's optional zip code dictionary you can lookup the city, county and state of a zip code using the functions listed in the table below.

Function	Reference Page	Description
<code>city(zip)</code>	Page 5101	This function looks up a zip code and returns the name of the city for that zip code. The zip code may be either a number or text. For example the formula <code>city(92831)</code> will return the city name <code>Fullerton</code> , while the formula <code>city("92648")</code> will return <code>Huntington Beach</code> . If there is more than one possible name, the function returns the primary zip code name as defined by the US Post Office. If the zip code is not a valid zip code the function will return an empty string (<code>""</code>). If the zip code dictionary has not been installed the function will return <code>--</code> .
<code>county(zip)</code>	Page 5132	This function looks up a zip code and returns the name of the county for that zip code. The zip code may be either a number or text. For example the formula <code>county(92831)</code> will return the county name <code>Orange</code> , while the formula <code>county("95234")</code> will return <code>San Joaquin</code> . If the zip code is not a valid zip code the function will return an empty string (<code>""</code>). If the zip code dictionary has not been installed the function will return <code>--</code> .
<code>state(zip)</code>	Page 5793	This function looks up a zip code and returns the two letter abbreviation for the name of the state the zip code is in. The zip code may be either a number or text. For example the formula <code>state(92831)</code> will return the state abbreviation <code>CA</code> , while the formula <code>state("15234")</code> will return <code>PA</code> . If the zip code dictionary has not been installed the function will return <code>--</code> .

For more information about purchasing this optional package please visit our web site, www.provue.com.

Note: If you have purchased Panorama's optional spelling dictionary, you can lookup a list of words using the `wordlist` statement in a procedure. See "[WORDLIST](#)" on page 5907 of the *Panorama Handbook* for details.

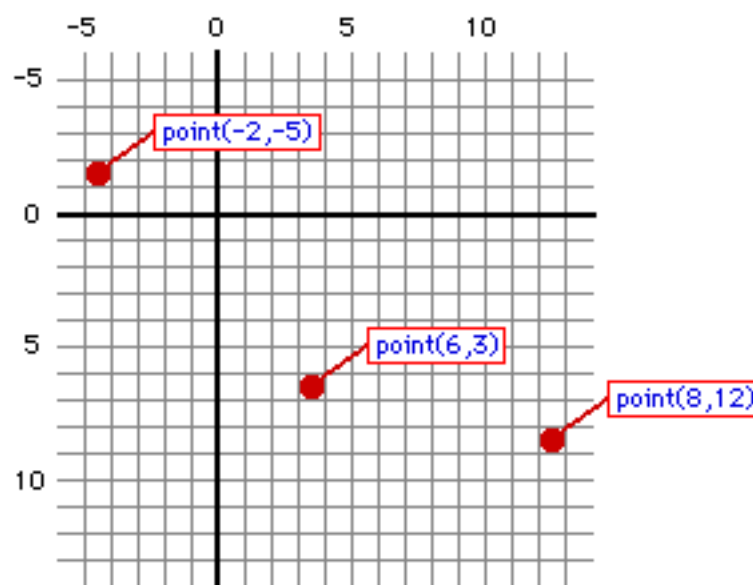
US Post Office Abbreviation Functions

These functions return text arrays that contain lists of official US Post Office abbreviations. These functions are designed to be used with the [arraylookup\(\)](#) and [arrayreverselookup\(\)](#) functions.

Function	Reference Page	Description
stateabbreviations()		This function returns a list of state abbreviations in this format: AL:ALABAMA ; AK:ALASKA ; ... This table is designed to be used with the arraylookup() and arraylookupreverse() functions.
statelookup(state)		This function looks up the name of a state from the state abbreviation (for example CALIFORNIA from CA). If the parameter does not match any state then the original value is returned.
uspssecondaryunits()		This function returns a list of USPS secondary suffix designation abbreviations in this format: APT:APARTMENT ; RM:ROOM ; ... This table is designed to be used with the arraylookup() and arraylookupreverse() functions.
uspsstreetsuffixes()		This function returns a list of USPS street suffix abbreviations in this format: ALY:ALLEY ; AVE:AVENUE ; ... This table is designed to be used with the arraylookup() and arraylookupreverse() functions.

Graphic Co-Ordinates

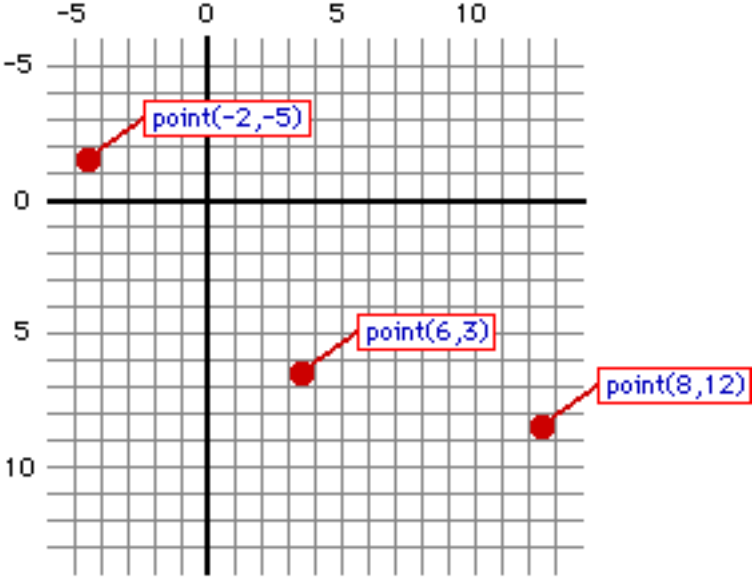
Many Panorama operations refer to locations on the screen, within a window, or within a form. The Macintosh uses an X-Y co-ordinate system to define locations. This X-Y system divides any area into an invisible grid of criss-crossing lines. There are 72 lines per inch. Each point where the lines intersect is identified by two numbers, the vertical and the horizontal position. The numbers increase as you move down and to the right. The illustration below shows a greatly expanded view of the X-Y coordinate system with several sample points.



See "[GRAPHIC COORDINATES](#)" on page 5330 of the *Panorama Handbook* for more information about coordinates.

Points

A point is a spot on the X-Y grid. A point has two elements, the vertical position and the horizontal position. Each position is a number (integer) between -32,767 and +32,767. A point combines these two numbers into a single number. Functions that work with points are listed in the table below.

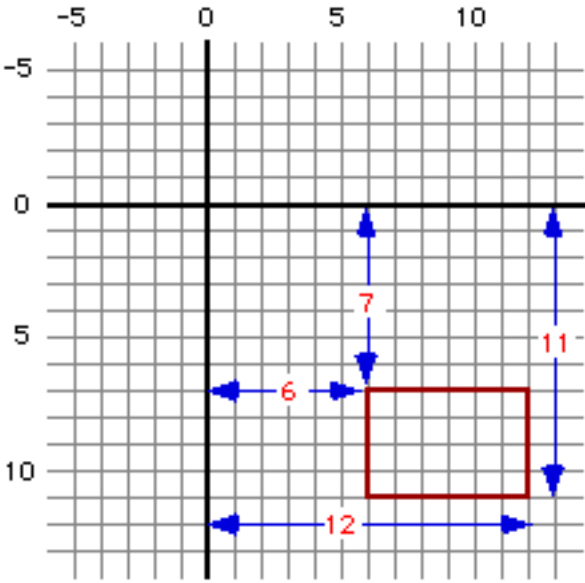
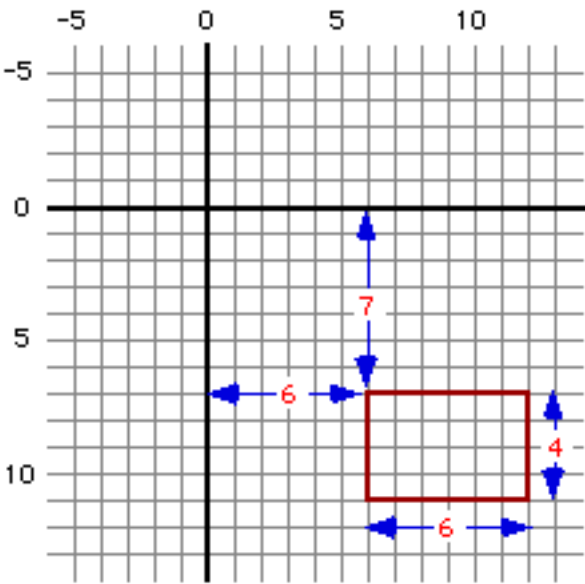
Function	Reference Page	Description
point(v,h)	Page 5606	<p>This function combines vertical and horizontal co-ordinates into a single number that describes the position of a point. V is the vertical position of the point. This must be a number between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.) H is the horizontal position of the point. This must be a number between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.) All dimensions are in pixels (1 pixel=1/72 inch).</p> <p>The function returns a number (an integer) that describes the location of the point. You can use this number in any function or statement that accepts a point as a parameter.</p> <p>The greatly magnified illustration below shows several sample points and the functions used to create them. Note that the actual point “hangs” down and to the right of the co-ordinate grid lines.</p> 
h(point)	Page 5338	<p>This function extracts the horizontal position from a point. The result is a number (an integer) that describes the horizontal position of the point. This number will be between -32,768 and +32,767. (Like standard cartesian co-ordinates, positive is right and negative is left.)</p>

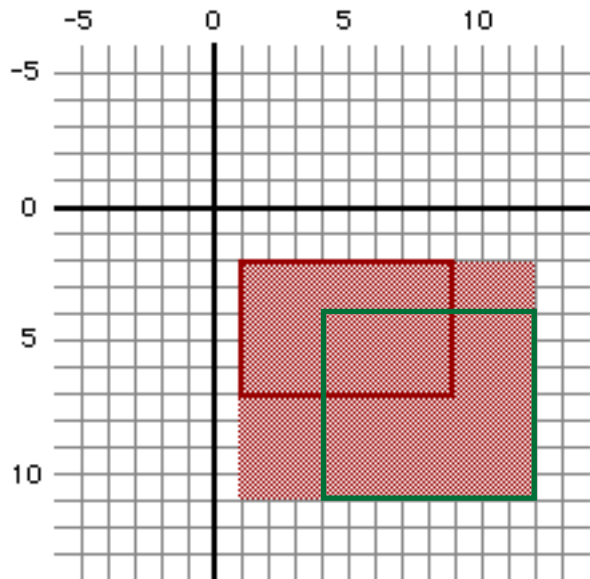
Function	Reference Page	Description
v(point)	Page 5885	This function extracts the vertical position from a point. The result is a number (an integer) that describes the vertical position of the point. This number will be between -32,768 and +32,767. (Unlike standard cartesian co-ordinates, positive is down and negative is up.)
info("click")	Page 5364	This function returns the location of the last mouse click in screen relative co-ordinates.
info("mouse")	Page 5398	This function returns the current location of the mouse in screen relative co-ordinates.
pointstr(point)		This function converts a point value into text in the format V,H (for example 34,56).
xytoxy(point,from,to)	Page 5910	<p>This function converts a point or rectangle from one co-ordinate system to another. There are three possible co-ordinate systems: Screen Relative, Window Relative, and Form Relative (see "GRAPHIC COORDINATES" on page 5330 of the <i>Panorama Handbook</i> for illustrations of these three systems).</p> <p>The function has three parameters: point, from and to. Point is the point or rectangle that you want to convert to another co-ordinate system. From is the current co-ordinate system for the point, while to is the new co-ordinate system. The three options for these parameters are:</p> <pre>"Screen" (may be abbreviated "S" or "s") "Window" (may be abbreviated "W" or "w") "Form" (may be abbreviated "F" or "f")</pre> <p>This function returns a new, translated point that has been converted to a different co-ordinate system (for example screen relative to window relative). (Note: This function can work with rectangles as well as points — see the next section.)</p>

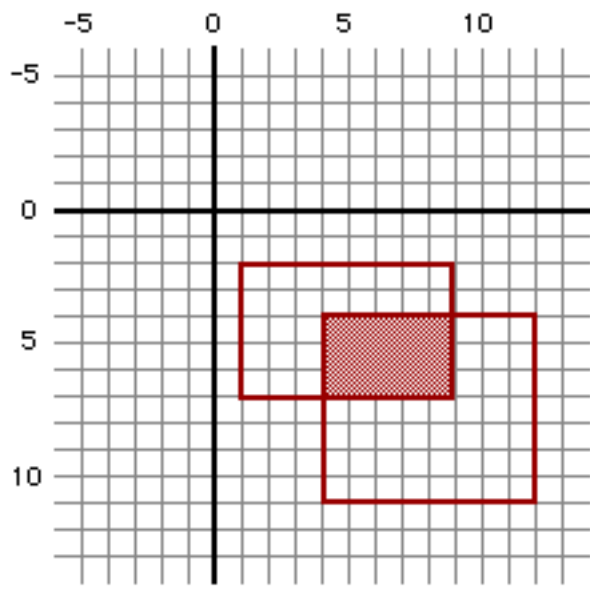
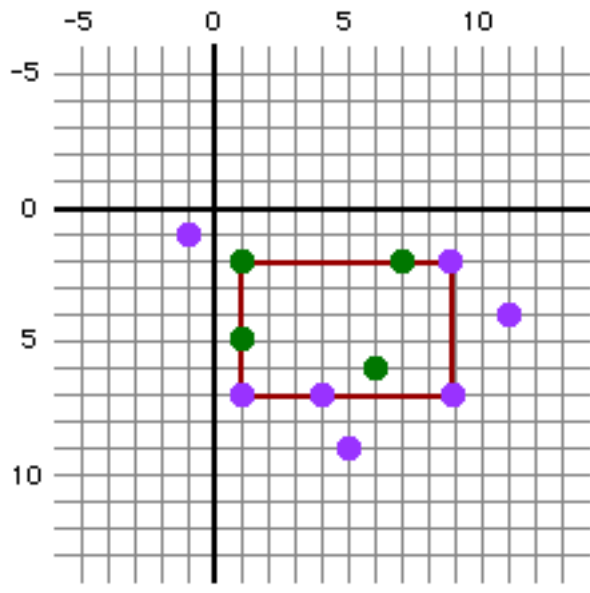
Rectangles

A rectangle is a simply box laid out on the X-Y grid. A rectangle is defined by four co-ordinates: top, left, bottom, right. The rectangle data type stores these four co-ordinates as raw binary data in an 8 byte text data item (see “[Raw Binary Data](#)” on page 156).

Don't confuse the rectangles described in this section with rectangles that are graphic objects in a form. The rectangle data type merely describes an imaginary rectangle on the X-Y grid. This may correspond to a rectangle that is actually visible, but not necessarily.

Function	Reference Page	Description
rectangle(top,left,bottom,right)	Page 5631	<p>This function defines a rectangle from four dimensions: top, left, bottom and right. These parameters specify the location of the edges of the rectangle. All measurements are in pixels (1 pixel = 1/72 inch). The formula below creates a rectangle that is 4 pixels high and 6 pixels wide.</p> <pre data-bbox="989 871 1371 907">rectangle(7,6,11,12)</pre> <p>Here is a magnified view of this rectangle.</p> 
rectanglesize(top,left,height,width)	Page 5636	<p>This function defines a rectangle from four dimensions: top, left, height and width. All measurements are in pixels (1 pixel = 1/72 inch). The height and width must be numbers between 0 and +32,767. The formula below creates a rectangle that is 4 pixels high and 6 pixels wide.</p> <pre data-bbox="989 1832 1410 1869">rectanglesize(7,6,4,6)</pre> <p>Here is a magnified view of this rectangle.</p> 

Function	Reference Page	Description
<code>rtop(rectangle)</code>	Page 5685	This function extracts the position of the top edge of a rectangle. The function returns a number between -32,768 and 32768. This is the position of the top edge of the rectangle (in pixels).
<code>rleft(rectangle)</code>	Page 5680	This function extracts the position of the left edge of a rectangle. The function returns a number between -32,768 and 32768. This is the position of the left edge of the rectangle (in pixels).
<code>rbottom(rectangle)</code>	Page 5630	This function extracts the position of the bottom edge of a rectangle. The function returns a number between -32,768 and 32768. This is the position of the bottom edge of the rectangle (in pixels).
<code>rright(rectangle)</code>	Page 5683	This function extracts the position of the right edge of a rectangle. The function returns a number between -32,768 and 32768. This is the position of the right edge of the rectangle (in pixels).
<code>rheight(rectangle)</code>	Page 5678	This function computes the height of a rectangle. The function returns a number between 0 and 65535. This value is the height of the rectangle (in pixels).
<code>rwidth(rectangle)</code>	Page 5688	This function computes the width of a rectangle. The function returns a number between 0 and 65535. This value is the width of the rectangle (in pixels).
<code>xytoxy(rect,from,to)</code>	Page 5910	<p>This function converts a point or rectangle from one co-ordinate system to another. There are three possible co-ordinate systems: Screen Relative, Window Relative, and Form Relative (see "GRAPHIC COORDINATES" on page 5330 of the <i>Panorama Reference</i> for illustrations of these three systems).</p> <p>The function has three parameters: rect, from and to. Rect is the point or rectangle that you want to convert to another co-ordinate system. From is the current co-ordinate system for the point, while to is the new co-ordinate system. The three options for these parameters are:</p> <pre>"Screen" (may be abbreviated "S" or "s") "Window" (may be abbreviated "W" or "w") "Form" (may be abbreviated "F" or "f")</pre> <p>This function returns a new, translated rectangle that has been converted to a different co-ordinate system (for example screen relative to window relative). (Note: This function can work with points as well as rectangles — see the previous section.)</p>
<code>unionrectangle(rect1,rect2)</code>	Page 5868	<p>This function defines a rectangle by combining two rectangles. The new rectangle is large enough to exactly cover both of the input rectangles. The illustration below shows how this function combines two rectangles, defining a third rectangle that covers the original two rectangles.</p> 






Function	Reference Page	Description
<p><code>intersectionrectangle(rect1,rect2)</code></p>	<p>Page 5459</p>	<p>This function defines a rectangle by combining two rectangles. The new rectangle is the area where the two rectangles overlap (if any). If the two rectangles do not touch each other the function will return an empty rectangle (same as <code>rectangle(0,0,0,0)</code>).</p> <p>The illustration below shows how this function combines two rectangles, creating a third rectangle where the original two rectangles overlap:</p> 
<p><code>inrectangle(point,rectangle)</code></p>	<p>Page 5453</p>	<p>This function checks to see if a point is inside a rectangle. There are two parameters: <code>point</code> and <code>rectangle</code>. The result is true or false. If the point is inside the rectangle, the function returns true (-1). If the point is not inside the rectangle, the function returns false (0). You can use this function with the <code>if</code> statement (see “IF Statements” on page 257) and the <code>?(</code> function (see “The ? Function” on page 130).</p> <p>The illustration below shows a rectangle and several points. Green points are inside the rectangle, purple points are not. Notice that points on the top and left edge of the rectangle are considered inside. Points on the bottom and right edge are considered outside.</p> 

Function	Reference Page	Description
rectanglecenter(largrect,smallrect)	Page 5635	<p>This function adjusts a small rectangle so that it is centered inside of a larger rectangle. Largerect is a large rectangle. How large is large? Well, it should at least be larger than the smallrect rectangle. Smallrect is a small rectangle. How small is small? Well, it should at least be small enough to fit inside the largerect rectangle, although the function will do its best to center it even if it does not fit.</p> <p>The formula below creates a 1 inch square rectangle that is centered within the current screen dimensions.</p> <pre>rectanglecenter(info("screenrectangle"), rectanglesize(0,0,72,72))</pre>
bestfitrectangle(lborder,rect)		<p>This function fits a rectangle inside a border. The original rectangle is enlarged or reduced as necessary to produce the best fit without changing the proportions. This function is useful for fitting an image (picture) inside a fixed size border.</p>
rectangleadjust(rect, Δtop, Δleft, Δbottom, Δright)	Page 5633	<p>This function adjusts all four edges of a rectangle independently. There are five parameters: rect, Δtop, Δleft, Δbottom and Δright. Rect is the original rectangle. The other four parameters are the distance each edge should be moved, which should be a number between -32,768 and +32,767.</p> <p>The formula below creates a rectangle that is inset 20 pixels from all four edges of the screen.</p> <pre>rectangleadjust(info("screenrectangle"),20,20,-20,-20)</pre> <p>The formula below creates a rectangle that is the same size as the button rectangle but shifted 1 inch (72 pixels) to the right.</p> <pre>rectangleadjust(info("buttonrectangle") ,0,72,0,72)</pre>
adjustxy(rectangle, boundary, deltav, deltah)	Page 5020	<p>This function adjusts the four corners of a rectangle. However, only corners that are inside a boundary are adjusted. Corners outside the boundary are left alone.</p> <p>There are four parameters: rectangle, boundary, deltav and deltah. Rectangle is the rectangle that is being adjusted. Boundary is a rectangle describing the area to be adjusted. Only points inside this rectangle will be adjusted. Deltav is the vertical distance each corner inside the boundary should be adjusted. Deltah is the horizontal distance each corner inside the boundary should be adjusted.</p>

Function	Reference Page	Description
<code>rectanglealign(lborder,rect)</code>		<p>This function aligns a small rectangle inside of a larger rectangle. The small rectangle can be aligned in one of nine positions: top left, top center, top right, left center, center, right center, bottom left, bottom center, bottom right. If an alignment axis is not specified it is assumed to be center, for example top is the same as top center. The order of the options does not matter, for example top left and left top are the same. If two conflicting specifications are made left will override right and top will override bottom.</p> <p>Here is an example that moves the current window to a top centered position on the main screen.</p> <pre>zoomwindowrectangle rectanglealign(info("screenrectangle"), info("windowrectangle"), "top center")</pre> <p>.</p>
<code>rectanglesizestr(rect)</code>		This function converts a rectangle into text in the format TOP,LEFT,HEIGHT,WIDTH (for example 100,20,80,30).
<code>rectanglestr(rect)</code>		Converts a rectangle into text in the format TOP,LEFT,BOTTOM,RIGHT (for example 100,20,180,50).
<code>info("buttonrectangle")</code>	Page 5361	This function returns a rectangle defining the edges of the button that was clicked on (needless to say, this function should be used in a procedure that is triggered by a button). The rectangle is in screen relative coordinates (use the <code>xytoxy()</code> function to convert to window or form relative co-ordinates).
<code>info("cursorrectangle")</code>	Page 5365	This function returns a rectangle defining the edges of the current data cell (if any). The rectangle is in screen relative coordinates (use the <code>xytoxy()</code> function to convert to window or form relative co-ordinates).
<code>info("screenrectangle")</code>	Page 5414	This function returns a rectangle defining the edges of the main screen (the screen that contains the menu bar). The rectangle is in screen relative coordinates.
<code>info("windowrectangle")</code>	Page 5446	This function returns a rectangle defining the edges of the current window. The rectangle is in screen relative coordinates.

Colors

We think of colors as the spectrum of the rainbow, but the computer builds up all colors from just three: red, green, and blue. By varying the relative intensity of these three colors the computer can generate all the colors of the rainbow. A Panorama color data item combines red, green and blue intensity values into a single data item. Color intensity is measured on a scale from 0 (completely dark) to 65,535 (full brightness). Values in between denote intermediate intensity. The table below shows a small sample of the colors that are possible.

RED	GREEN	BLUE	COLOR	SAMPLE
0	0	0	Black	
65535	65535	65535	White	
15000	15000	15000	Dark Gray	
45000	45000	45000	Light Gray	
65535	0	0	Pure Red	
0	65535	0	Pure Green	
0	0	65535	Pure Blue	
65535	0	65535	Purple	
65535	65535	0	Yellow	
0	65535	65535	Cyan	
3441	4276	32336	Dark Blue	
39235	30211	30211	Brown	
24367	23356	31931	Light Green	
65535	23356	2936	Orange	

Another way to specify a color is the **HSB**, or **Hue, Saturation, Brightness** system. Like the RGB system, the HSB system uses three numbers from 0 to 65,535 to describe a color. However, the three components have different meanings in this system.

The **Hue** component specifies where this color falls in the spectrum. If you are familiar with the standard Apple color picker, the Hue would specify the angle of the color from the center of the wheel.

The **Saturation** component refers to how intense this color is. Is it a very intense deep color, or is it a soft pastel color, or somewhere in between? Again using the standard Apple color picker, the Saturation would specify the distance of the color from the center of the wheel.

The **Brightness** component refers to how light or dark the color is. Is the color very bright, or is it almost black? This sounds similar to Saturation, but it isn't. Imagine a blue ball under a white light. As the light gets dimmer, the Hue and Saturation of the color don't change, but the Brightness does.

A color in a field or variable is just a piece of data that describes a color...you can't actually see the color. However, some SuperObjects allow you to control their color using a color data item, and you can look at or modify the color of any graphic object in a form using the functions and statements listed below.

Function	Reference Page	Description
<code>rgb(red,blue,green)</code>	Page 5677	This function creates a color by combining red, green, and blue primary colors. Red is the intensity of the red component of this color. This must be a number from 0 (black) to 65535 (full intensity). Green is the intensity of the green component of this color. This must be a number from 0 (black) to 65535 (full intensity). Blue is the intensity of the blue component of this color. This must be a number from 0 (black) to 65535 (full intensity).
<code>hsb(hue,saturation,brightness)</code>	Page 5347	This function creates a color by combining hue, saturation, and brightness components. Hue specifies where this color falls in the spectrum. This must be a number from 0 to 65535. Saturation specifies how intense this color is. Is it a very intense deep color, or is it a soft pastel color, or somewhere in between? This must be a number from 0 (black) to 65535 (full intensity). Brightness specifies how light or dark the color is. Is the color very bright, or is it almost black? This sounds similar to Saturation, but it isn't. Imagine a blue ball under a white light. As the light gets dimmer, the Hue and Saturation of the color don't change, but the Brightness does. This must be a number from 0 (black) to 65535 (full intensity).
<code>getwebcolor(webcolor,default)</code>		<p>This function will calculate an RGB color from a Netscape style web color. The first parameter specifies an HTML color, either in RRGGBB format (or #RRGGBB) or a named color (see list below). The second parameter allows you to specify what color to use if the first parameter is not recognized (for example a named color not on the list).</p> <p>List of named web colors: aliceblue, antiquewhite, aqua, aquamarine, azure, beige, bisque, black, blanchedalmond, blue, blueviolet, brown, burlywood, cadetblue, chartreuse, chocolate, coral, cornflowerblue, cornsilk, crimson, cyan, darkblue, darkcyan, darkgoldenrod, darkgray, darkgreen, darkkhaki, darkmagenta, darkolivegreen, darkorange, darkorchid, darkred, darksalmon, darkseagreen, darkslateblue, darkslategray, darkturquoise, darkviolet, deeppink, deepskyblue, dimgray, dodgerblue, firebrick, floralwhite, forestgreen, fuchsia, gainsboro, ghostwhite, gold, goldenrod, gray, green, greenyellow, honeydew, hotpink, indianred, indigo, ivory, khaki, lavender, lavenderblush, lawngreen, lemonchiffon, lightblue, lightcoral, lightcyan, lightgoldenrodyellow, lightgreen, lightgrey, lightpink, lightsalmon, lightseagreen, lightskyblue, lightslategray, lightsteelblue, lightyellow, lime, limegreen, linen, magenta, maroon, medianaquamarine, mediumblue, mediumorchid, mediumpurple, mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise, mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, navajowhite, navy, oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen, paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, powderblue, purple, red, rosybrown, royalblue, saddlebrown, salmon, sandybrown, seagreen, seashell, sienna, silver, skyblue, slateblue, slategray, snow, springgreen, steelblue, tan, teal, thistle, tomato, turquoise, violet, wheat, white, whitesmoke, yellow, yellowgreen</p>
<code>red(color)</code>	Page 5638	<p>This function extracts the red intensity from a color. This intensity is a number between 0 (black) and 65535 (full intensity). The example below calculates the red intensity of the color (in percent, from 0 to 100%).</p> <pre>red(HighlightColor)*100/65535</pre>

Function	Reference Page	Description
green(color)	Page 5333	This function extracts the green intensity from a color. This intensity is a number between 0 (black) and 65535 (full intensity). The example below calculates the green intensity of the color (in percent, from 0 to 100%). <code>green(HighlightColor)*100/65535</code>
blue(color)	Page 5076	This function extracts the blue intensity from a color. This intensity is a number between 0 (black) and 65535 (full intensity). The example below calculates the blue intensity of the color (in percent, from 0 to 100%). <code>blue(HighlightColor)*100/65535</code>
hue(color)	Page 5347	This function extracts the hue value from a color. Hue specifies where this color falls in the spectrum. This is a number from 0 to 65535.
saturation(color)	Page 5691	This function extracts the saturation intensity from a color. Saturation specifies how intense this color is. Is it a very intense deep color, or is it a soft pastel color, or somewhere in between? This is a number from 0 (black) to 65535 (full saturation).
brightness(color)	Page 5077	This function extracts the brightness of a color. Brightness specifies how light or dark the color is. Is the color very bright, or is it almost black? This sounds similar to Saturation, but it isn't. Imagine a blue ball under a white light. As the light gets dimmer, the Hue and Saturation of the color don't change, but the Brightness does. The brightness value is a number from 0 (black) to 65535 (full brightness).
info("formcolor")	Page 5378	This function returns the background color of the current form. If the current window does not contain a form, the function will return empty text ("").
black()		This function returns black (it is equivalent to <code>rgb(0,0,0)</code>).
gray(saturation)		This function returns a gray color. The saturation value is between 0 (white) and 100 (black).
htmlrgb(string)		This function converts text formatted as an HTML color (for example <code>FFFFFF</code> for white or <code>FF0000</code> for red) into a Panorama color value.
htmlrgbstr(color)		This function converts a Panorama color value into text formatted as an HTML color (for example <code>BB4DFD</code>).
white()		This function returns white (it is equivalent to <code>rgb(65535,65535,65535)</code>).

If you are writing a procedure there are also two procedure statements that deal with color. The `colorwheel` statement (see "[COLORWHEEL](#)" on page 5119 of the *Panorama Reference*) opens a dialog for picking a color. The `formcolor` statement (see "[FORMCOLOR](#)" on page 5262 of the *Panorama Reference*) changes the background color of the current form.

Raw Binary Data

At the core, computers work with 1's and 0's, on and off, true and false. This is called binary data, because there are only two options. Fortunately, Panorama users don't ever have to deal with raw binary data. The programmers take the 1's and 0's and give them structure to create text, numbers, pictures, and other complex elements.

It's not much fun, and it's rarely necessary, but Panorama does allow a programmer to work with raw, unstructured, binary data: 1's and 0's. When you work with raw binary data it will always be in a text field or variable. Panorama normally interprets text as a series of characters. The functions listed in this section, however, do not interpret the binary data as characters. Instead, they allow you to directly access and manipulate the 1's and 0's. Panorama uses the text data type to hold raw binary data because text may be of any length and places no restrictions on the binary information that is placed in it. (However, the text may look very strange if you display it in the data sheet or on a form.)

Note: Panorama never requires you to use raw binary data. However, raw binary data may be useful for working with data from the operating system or with data generated by another program (or to generate data for another program). If you don't already know you need to use raw binary data, you can probably skip this section (unless you are curious). If you are a C, Java or Pascal programmer, these functions let you work with virtually any data structure you can cook up.

The functions that work with binary data are listed in the table below.

Function	Reference Page	Description
byte(number)	Page 5079	This function converts a number into a single byte of binary data. (Note: the byte(function is basically the same as the chr(function.) The number parameter must be between 0 and 255. This function converts the number into a single byte of binary data (8 bits).
word(number)	Page 5906	This function converts a number into a single word (2 bytes) of binary data. Number is the value that you want to convert into a binary number. This value must be between 0 and 65,535. This function converts the number into a two bytes of binary data (16 bits).
longword(number)	Page 5499	This function converts a number into a single longword (4 bytes) of binary data. Number is the value that you want to convert into a binary number. This value must be an integer. This function converts the number into a four bytes of binary data (32 bits).
binaryvalue(data)	Page 5075	This function converts binary data (a byte, word, or longword) into a number. Data is the binary value that you want to convert into a number. This value must be a byte, a word (2 bytes) or a longword (4 bytes). The result is an integer.

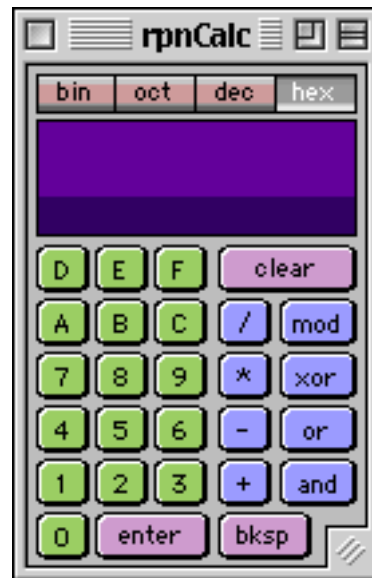
Function	Reference Page	Description
textstuff(text,new,position)	Page 5853	<p>This function replaces one or more characters in the middle of a piece of text. Text is the original text data item that contains one or more characters you want to replace. New is the new text that you want to use to replace characters in the original text data item. Position is the location within the original text where you want to replace text. The position is a number starting with zero. The function returns a copy of the original text item with one or more characters replaced.</p> <p>This example replaces two characters in a 24 character text item.</p> <pre>textstuff("Temperature: 87 degrees"," 92",13)</pre> <p>The operation of this formula is shown in the table below.</p> <pre>Original Text: Temperature: 87 degrees New Text: 92 Position: 13 ----- Result (TEMP): Temperature: 92 degrees</pre> <p>If the new text is positioned beyond the end of the original text, the characters in between are undefined.</p> <pre>textstuff("Temp:"," 92",13)</pre> <p>The operation of this formula is shown in the table below.</p> <pre>Original Text: Temp: New Text: 92 Position: 13 -----x----- Result (TEMP): Temp: \#ø•°çΠf92</pre> <p>The characters in between (in orange) may be anything. (Of course you could use another textstuff(function to fill them in, or you could add characters to the original text before using textstuff(in the first place.)</p>
string255(text,space)	Page 5800	<p>This function converts text into a Pascal string. A Pascal String is a special text format that is sometimes used by the Macintosh ROM's (also called a String255 or Str255 because the text is limited to a maximum length of 255 characters). (Pascal is the name of a computer language, which in turn is named after a famous mathematician.)</p> <p>The function has two parameters: text and space. Text is the text that you want to convert into a Pascal string. This text should be less than 255 characters long. Space is a number defining the amount of space taken up by the Pascal string. If space is zero, the string may be up to 255 characters, and is not padded. If space is from 1 to 255, the string255(function makes sure that the string takes up exactly this amount of space. If the string is too long, it will be cut off. If the string is too short, it will be padded with nulls (bytes containing zeroes).</p> <p>The function returns a text data item containing a Pascal string.</p>

Function	Reference Page	Description
text255(data)	Page 5851	This function converts binary data containing a Pascal String into regular text. A Pascal String is a special text format that is sometimes used by the Macintosh ROM's (also called a String255 or Str255 because the text is limited to a maximum length of 255 characters). (Pascal is the name of a computer language, which in turn is named after a famous mathematician.) The function returns the text equivalent of the Pascal String passed to it.
radix(radix,text)	Page 5626	<p>This function converts a text item containing a hex, octal, or binary number into a standard Panorama number (decimal). See “NON DECIMAL NUMBERS” on page 5543 of the <i>Panorama Reference</i> for background information on hex, octal and binary numbers. Radix is the base for the numbering system you are converting from. Legal radix values are 2, 4, 8, 16 or 32. Or you can specify the radix as "binary" (same as 2), "octal" (same as 8) or "hex" (short for hexadecimal, same as 16). Text is a text item that contains the non-decimal number you want to convert. This function normally returns an integer that contains the decimal (base 10) number corresponding to the hex, octal, or binary number input to the function.</p> <p>If the radix is hex and there are more than 8 digits in the input text, or if the radix is binary and there are more than 32 digits, this function will return a raw binary value instead of a number. This binary value may be of unlimited length. Like all binary values, it cannot be calculated with, but should be handled as a text item.</p>
radixstr(radix,number)	Page 5628	<p>This function converts a number into a text item containing the equivalent hex, octal, or binary number. See “NON DECIMAL NUMBERS” on page 5543 of the <i>Panorama Reference</i> for background information on hex, octal and binary numbers. Radix is the base for the numbering system you are converting from. Legal radix values are 2, 4, 8, 16 or 32. Or you can specify the radix as "binary" (same as 2), "octal" (same as 8) or "hex" (short for hexadecimal, same as 16). Number is the number you want to convert to hex, octal, or binary. If the radix is 2, 16, "binary", or "hex" the number can be a raw binary data (text) value. This function returns a text item that contains the hex, octal, or binary number equivalent to the number (or binary data) passed to the function. The first example converts the decimal value 256 to hexadecimal.</p> <pre>radixstr(16,256)</pre> <p>This function will calculate that 256₁₀ is 100 hex.</p> <p>Here is another example:</p> <pre>radixstr("binary",5)</pre> <p>This will calculate that 5₁₀ is 00000000000000000000000000000000101 binary.</p>
bit(number)		This function converts a bit number (1 to 32) into a number (1, 2, 4, 8, 16, etc.)
bytearray(data,index)		This function extracts a value from an array of bytes. This is not a Panorama style delimited array but a C style array of 8 bit values. The result is an integer.
chararray(data,index)		This function extracts a characters from an array of characters. This is not a Panorama style delimited array but a C style array of ASCII characters. The result is an single characters.
chunkarray(data,index,chunklength)		This function extracts a binary chunk from an array of chunks. This is not a Panorama style delimited array but a C style array of binary chunks. The result is a binary value (text).

Function	Reference Page	Description
<code>getbit(number,bitnumber)</code>		This function returns a true or false value by testing a bit. The bit number may be from 1 to 32. If the bit is set, the value will be true, if it is not set, the value will be false.
<code>hex(string)</code>		Converts text with hex characters into a number. For example the value of <code>hex("0C2")</code> is 194.
<code>hexstr(number)</code>		Converts a number into text in hex format. For example the value of <code>hexstr(194)</code> is "000000C2".
<code>hexbyte(number)</code>		Converts a 8-bit number into text in hex format. For example the value of <code>hexstr(194)</code> is "C2".
<code>hexlong(number)</code>		Converts a 32-bit number into text in hex format. For example the value of <code>hexstr(194)</code> is "000000C2".
<code>hexword(number)</code>		Converts a 16 bit number into text in hex format. For example the value of <code>hexstr(194)</code> is "00C2".
<code>longwordarray(data,index)</code>		This function extracts a value from an array of longwords. This is not a Panorama style delimited array but a C style array of 32 bit values. The result is an integer.
<code>onescomplement(number)</code>		This function returns the one's complement of a 32 bit number (all bits are reversed)
<code>setbit(number,bitnumber,truefalse)</code>		This function sets one bit within a number, without disturbing any of the other bits. The bit number may be from 1 to 32. The bit will be set based on the true/false parameter - set if true, cleared if false.
<code>wordarray(data,index)</code>		This function extracts a value from an array of words. This is not a Panorama style delimited array but a C style array of 16 bit values. The result is an integer.
<code>encodebase64(data,linelength)</code>		This function encodes text using the Base64 algorithm. Base64 encoding is widely used on the web and in e-mail for encoding binary data and allowing it to be transmitted as plain text. For more information on Base64 encoding see http://en.wikipedia.org/wiki/Base64 . The data parameter is the data to be encoded. LineLength is the maximum line length of the encoded text. A common value is around 70 characters per line.
<code>decodebase64(data)</code>		This function decodes text that has been encoded using Base64 encoding. It is the reverse of the <code>encodebase64()</code> function above. If you first use <code>encodebase64()</code> then <code>decodebase64()</code> you will get back the original data..

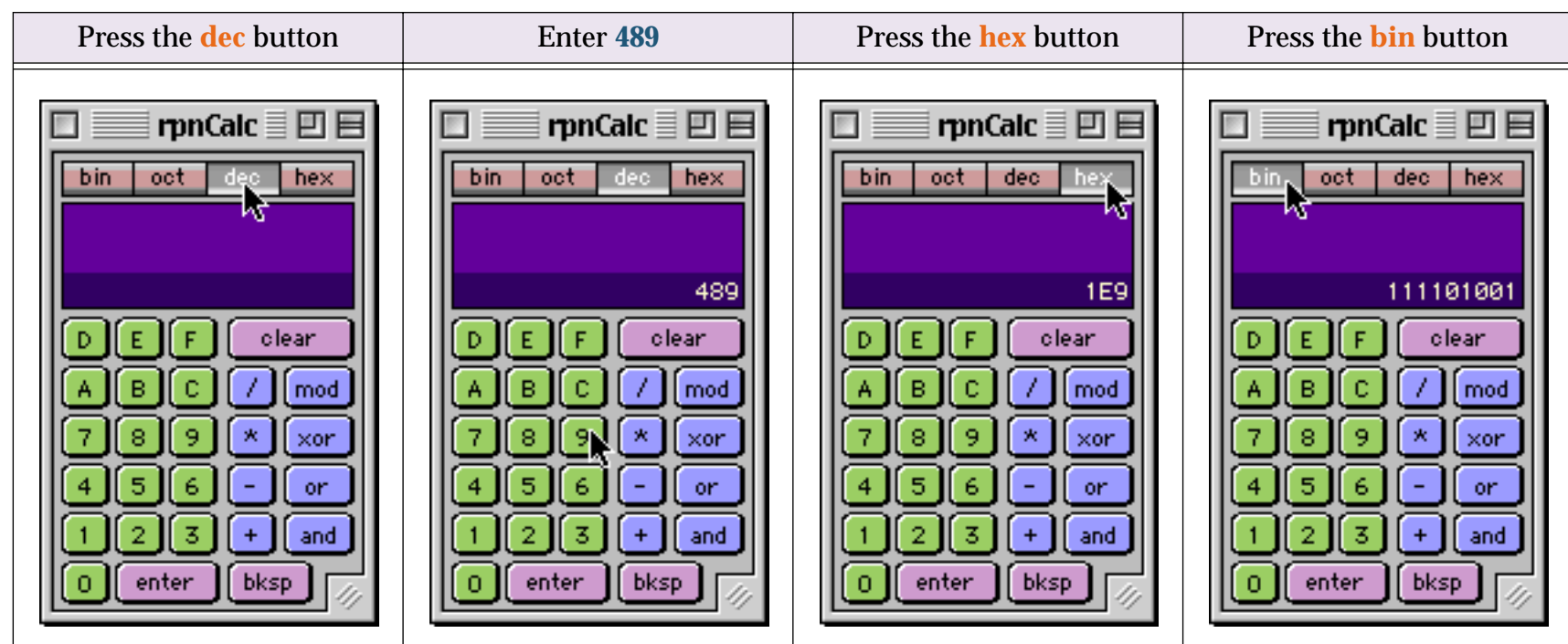
The RPN Programmer's Calculator

Panorama includes a built in calculator that can perform calculations in decimal (base 10), hexadecimal (base 16), octal (base 8) and binary (base 2). To open this calculator choose the **RPN Programmer's Calculator** from the Wizard menu.



Converting Between Different Bases

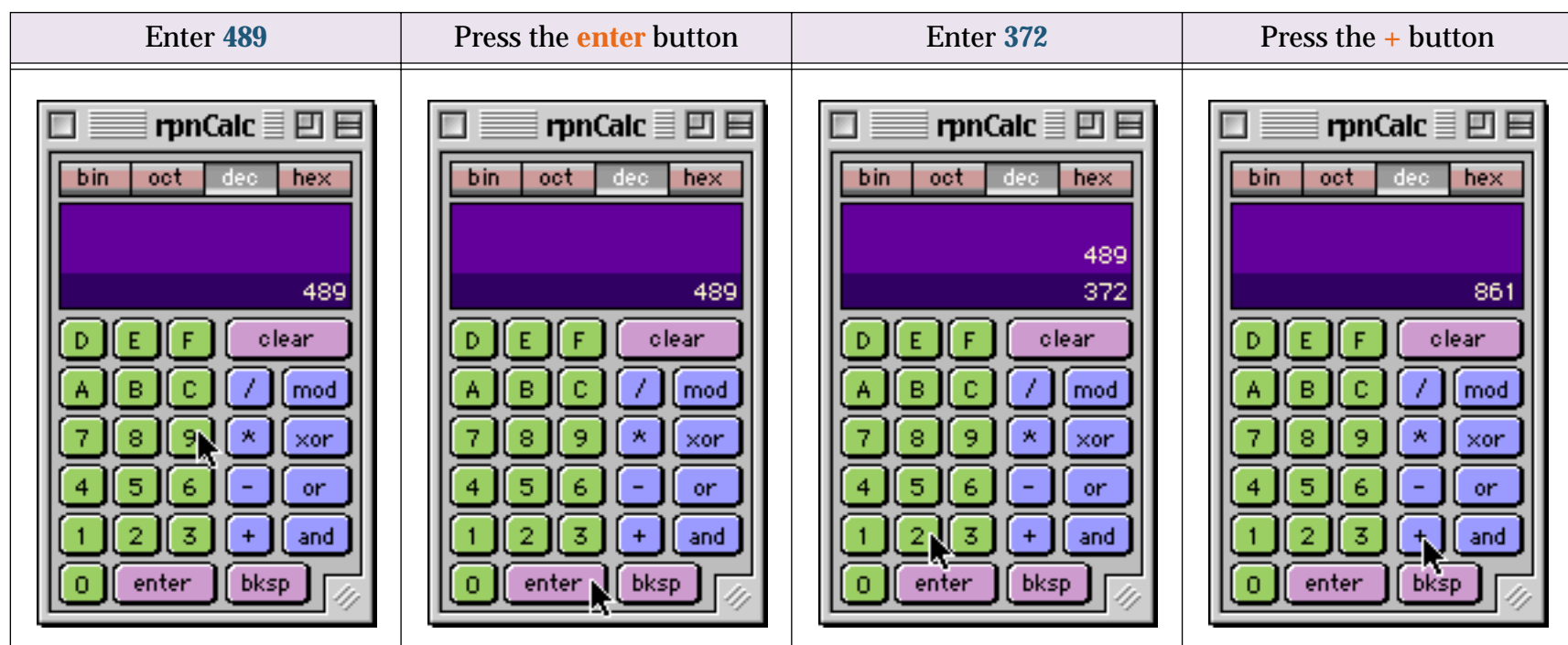
The calculator can be used to convert numbers from decimal to hex, hex to decimal, or in fact from any of the four supported bases into any other. For example, suppose you want to convert the number **489** (decimal) into hex. Start by pressing the **dec** (for decimal) button, then enter **489**, then press the **hex** button. The result is **1E9** (remember, hexadecimal numbers may include the digits **A-F** in addition to **0-9**). If you wanted to see the same number in binary you would press the **bin** button. You can change the number base at any time.



Calculations with Reverse Polish Notation

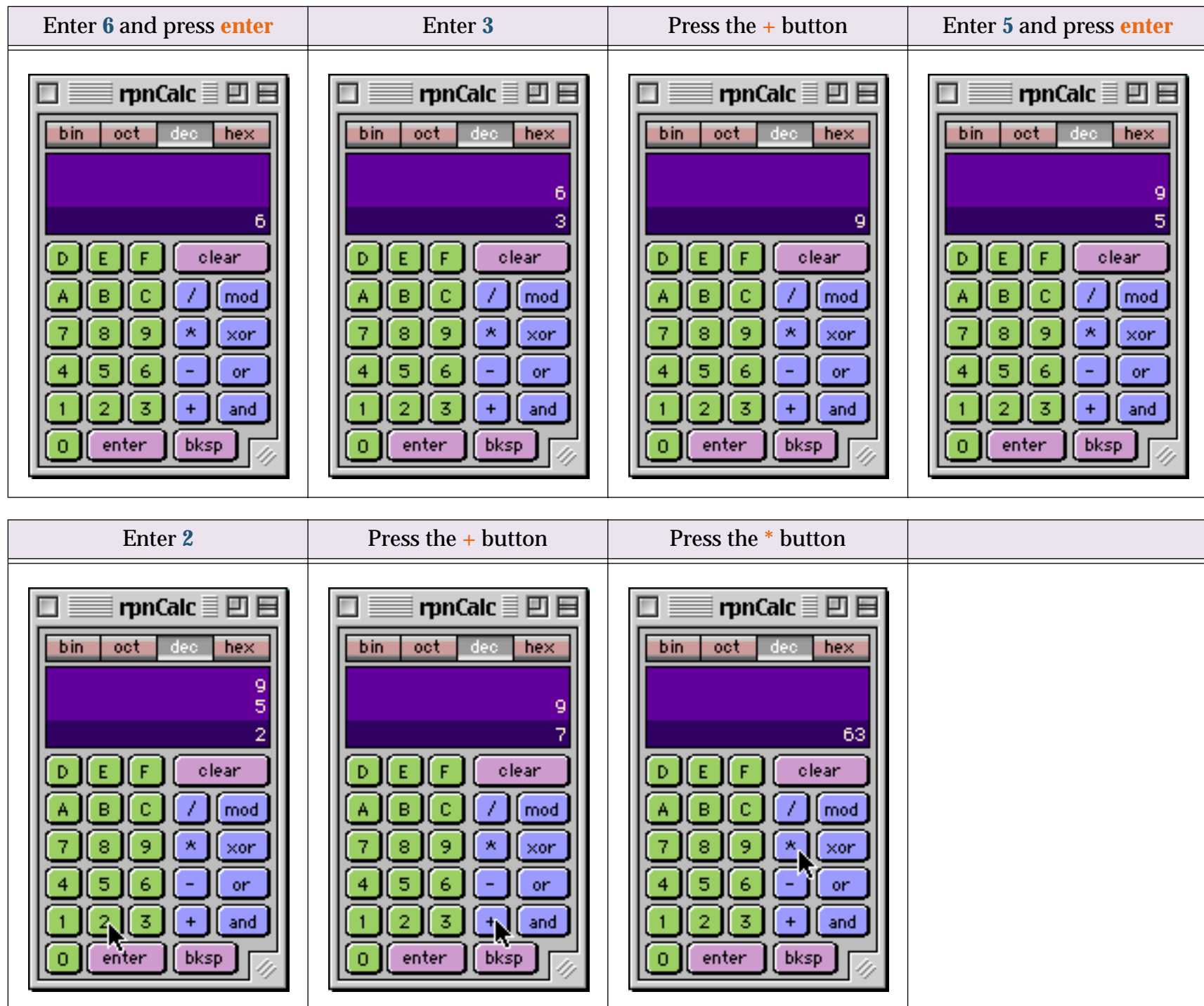
RPN is a variation on a parentheses-free mathematical logic known as "Polish Notation," developed by the Polish logician Jan Lukasiewicz (1878-1956). Over the years the most popular calculators in business and science have been HP calculators. These feature Reverse Polish Notation, especially the HP-12C business calculators and the HP-41 series. Texas Instruments was the other big player in calculators, and their machines used algebraic notation. You were either an RPN supporter, walking around with the motto **ENTER > = (RPN is greater than Algebraic)** emblazoned on T-shirts, or you were a TI algebraic "heretic." Here at ProVUE we were weaned on the original HP-35 scientific calculator in the late 70's, so our programmer's calculator uses Reverse Polish Notation.

With RPN, numbers are entered or "stacked" in the register. RPN is implemented by means of a numeric stack, the enter key, and the convention of "postfix operators." Postfix operators simply means that the user specifies the operation to be performed after the entry of numbers, instead of in the middle. For example, suppose you wanted to add $489+372$. Here's how you do this with the RPN calculator.



According to HP, "RPN is an effective way to deal with arithmetic expressions in programming. RPN makes it possible to perform compound calculations with a minimum of special symbols and no punctuation. Numbers are stored in the register. With the elimination of parentheses and the consistency of the entry method, the calculator accepts more of the problem-solving burden, reducing the user's time and effort."

The RPN system eliminates the need for parentheses. Here's how you would calculate $(6+3)*(5+2)$.



Performing this calculation in RPN requires only 9 keystrokes, vs. 11 using conventional algebraic notation. Ok, so it's a religious issue. So sue me (and all my RPN loving cronies)!

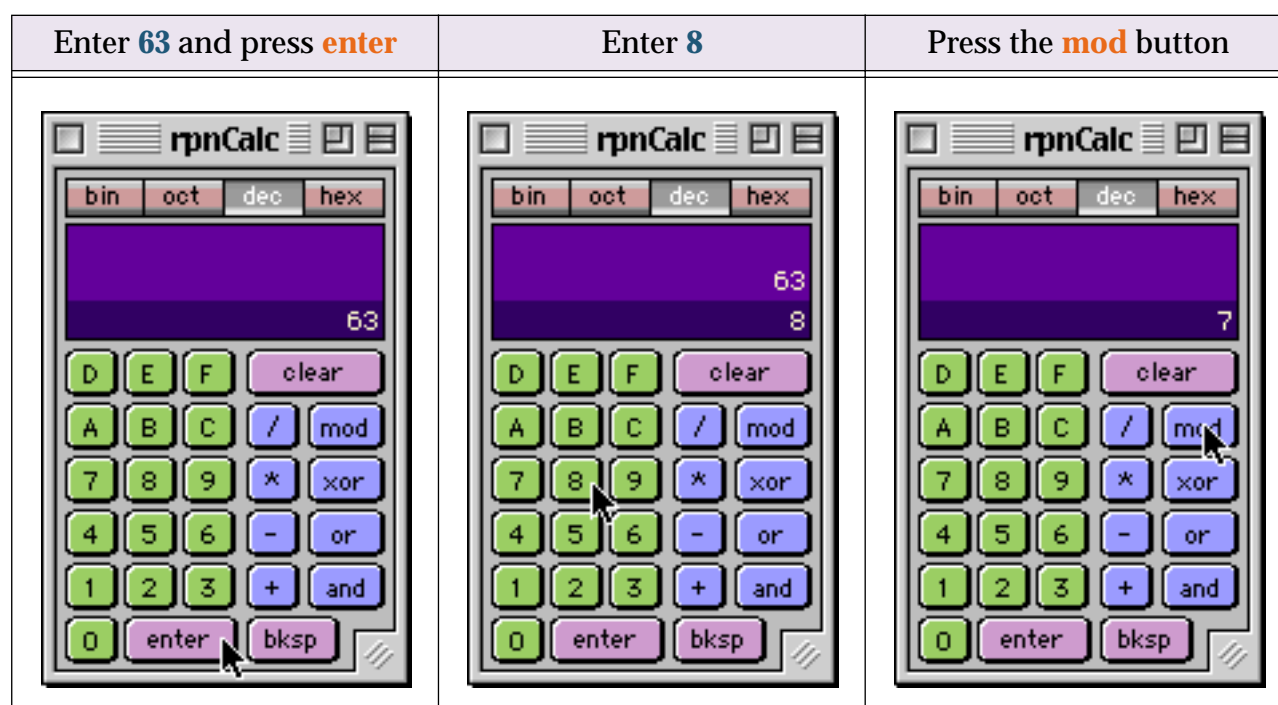
The purple area shows the numeric stack, where intermediate results are stored. You can expand the calculator window to see more entries in this numeric stack.

Boolean Operators

The four buttons on the right (**mod**, **xor**, **or** and **and**) are the boolean functions. These combine two numbers on a bit by bit basis. The table below explains each of these functions. The example numbers (**A**, **B** and **Result**) are in binary, but these operators will work in any number base..

Operator	Description	A	B	Result
and	For each bit in A and B, the result will be 1 if both A and B are 1. It will be 0 if either A or B are zero.	10110	00101	00100
or	For each bit in A and B, the result will be 1 if either A or B is 1. It will be 0 if both A and B are zero.	10110	00101	10111
xor	For each bit in A and B, the result will be 1 if A and B are different. It will be 0 if both A and B are the same.	10110	00101	10011
mod	This operator computes the remainder when dividing A by B	10110	00101	10

This example shows how to calculate the remainder when dividing **63** by **8**.



The remainder is **7**.

Disk Files and Folders

Panorama formulas can directly access files and directories on the disk. The functions below read information from the disk. For a more detailed discussion of file i/o, including writing to files, see “[Directly Reading and Writing Disk Files](#)” on page 421.

Function	Reference Page	Description
folder(path)	Page 5259	<p>This function creates a binary data item that unambiguously describes the location of a folder on the hard disk. This pathid can be used in other functions and statements. Path is a complete description of the path to this folder. On the Macintosh a path looks like this:</p> <pre>My Disk:System Folder:Extensions:</pre> <p>On a Windows computer a path looks like this:</p> <pre>C:\Windows\Temporary</pre> <p>This function returns a 6 byte binary data item that unambiguously describes the location of the folder. However, if the folder does not exist the function returns an empty binary data item ("").</p>
folderpath(pathid)	Page 5260	<p>This function takes a six byte pathid (see the folder(function above) and converts it to a textual description of the path to that folder. A pathid is a binary data item that unambiguously describes the location of a folder on the hard disk. Pathid's are created by the folder(and dbinfo(functions, and the openfiledialog and savefiledialog statements.</p> <p>The function returns complete a description of the path to this folder. On the Macintosh a path looks like this:</p> <pre>My Disk:System Folder:Extensions:</pre> <p>On a Windows computer a path looks like this:</p> <pre>C:\Windows\Temporary</pre>
filefolder(text)		<p>This function returns the folder ID from a combined path and filename. For example the function <code>filefolder("Alaska:Denali:Image45.jpg")</code> would return the folder id for the path "Alaska:Denali:".</p>

Function	Reference Page	Description
fileload(folder,file)	Page 5231	<p>This function reads the entire contents of any file on disk. It is especially useful for reading text files. Folder is a pathid that unambiguously describes the location of the folder. A pathid is a binary data item that unambiguously describes the location of a folder on the hard disk. pathid's are created by the folder(and dbinfo(functions, and the openfiledialog and savefiledialog statements. If this parameter is empty text ("" the folder containing the current database is assumed. File is the name of the file that is to be read.</p> <p>This function returns the entire contents of the file as an item of text. (Technical note: Macintosh files may be split up into two components, called the “data fork” and the “resource fork.” The fileload(function reads the data fork, but not the resource fork. You can read the resource fork by using a special statement, see “Reading and Writing Resource Forks” on page 428.)</p> <p>This example reads a file named Report.txt. This file must be in the same folder as the current database.</p> <pre>fileload("", "Report.txt")</pre> <p>The example below reads the contents of the Macintosh notebook file.</p> <pre>fileload(info("systemfolder"), "Note Pad File")</pre> <p>For very large files you may need to read in only a portion at a time using the fileloadpartial(function (see below).</p>
fileloadpartial(folder,file,start,len)	Page 5233	<p>This function reads a portion of the contents of any file on disk. It is especially useful for reading text files. Folder is a pathid that unambiguously describes the location of the folder. A pathid is a binary data item that unambiguously describes the location of a folder on the hard disk. pathid's are created by the folder(and dbinfo(functions, and the openfiledialog and savefiledialog statements. If this parameter is empty text ("" the folder containing the current database is assumed. File is the name of the file that is to be read. Start is the first byte (character) of the file that should be read. This function assumes bytes in the file are numbered starting from 0. Len is the number of bytes that should be read.</p> <p>The function returns a portion of the contents of the file as an item of text. (Technical note: Macintosh files may be split up into two components, called the “data fork” and the “resource fork.” The fileloadpartial(function reads the data fork, but not the resource fork. You can read the resource fork by using a special statement, see “Reading and Writing Resource Forks” on page 428.)</p>
filesize(folder,file)	Page 5238	<p>This function determines the size of any file on disk. Folder is a pathid that unambiguously describes the location of the folder. A pathid is a binary data item that unambiguously describes the location of a folder on the hard disk. pathid's are created by the folder(and dbinfo(functions, and the openfiledialog and savefiledialog statements. If this parameter is empty text ("" the folder containing the current database is assumed. File is the name of the file that is to be measured.</p> <p>This function returns a number—the size of the entire contents of the file. (Technical note: Macintosh files may be split up into two components, called the “data fork” and the “resource fork.” The filesize(function reads the size of the data fork, but not the resource fork.)</p> <p>The example below determines the size of the Macintosh notebook file.</p> <pre>filesize(info("systemfolder"), "Note Pad File")</pre>

Function	Reference Page	Description
fileinfo(folder,file)	Page 5229	<p>This function gets information about a file (or folder) on the disk, including the size, creation and modification date and time, type, creator and lock status. Folder is a pathid that unambiguously describes the location of the folder. A pathid is a binary data item that unambiguously describes the location of a folder on the hard disk. pathid's are created by the folder(and dbinfoc(functions, and the openfiledialog and savefiledialog statements. If this parameter is empty text ("") the folder containing the current database is assumed. File is the name of the file that you are requesting information about.</p> <p>This function returns a text array with 8 elements separated by carriage returns. (However, if the specified file does not exist it returns empty text ("")). The eight elements are:</p> <ol style="list-style-type: none"> 1) Type of item. This is either "File" or "Folder" 2) Type (4 bytes) and Creator (4 bytes). Here are some typical Type/Creator values. <ul style="list-style-type: none"> ZEPDKASX Panorama database KSETKASX Panorama file set APPLKASX Panorama application CWWPBOBO ClarisWorks word processing file STAKWILD Hypercard Stack TIFF8BIM TIFF file (Photoshop) EPSFART3 Adobe Illustrator (version 3) <p>This is only a small sample of the types and creators you will find on your hard disk.</p> <ol style="list-style-type: none"> 3) Creation Date in internal Panorama format. Although this is a number, it has been converted to text. If you convert the number back to text you can format the date with datepattern(. 4) Creation Time in seconds since midnight. Although this is a number, it has been converted to text. If you convert the number back to text you can format the time with timepattern(. 5) Modification Date in internal Panorama format. Although this is a number, it has been converted to text. If you convert the number back to text you can format the date with datepattern(. 6) Modification Time in seconds since midnight. Although this is a number, it has been converted to text. If you convert the number back to text you can format the time with timepattern(. 7) File size in bytes. (Or if the specified file is actually a directory, this is the number of files in directory 8) File status: This is either "Locked" or "Unlocked".
filesuperdate(folder,file)		<p>This function returns the modification date and time of a file as a superdate. For example this can be handy for comparing to see which of two files is newer.</p>

Function	Reference Page	Description
listfiles(folder,filter)	Page 5476	<p>This function builds a text array listing the files in a folder. Folder is a pathid that unambiguously describes the location of the folder. A pathid is a binary data item that unambiguously describes the location of a folder on the hard disk. pathid's are created by the folder(and dbinfo(functions, and the openfiledialog and savefiledialog statements. If this parameter is empty text ("") the folder containing the current database is assumed. Filter is a text item that specifies what type (or types) of files (and folders) to list. If this is an empty text item ("") all files will be listed. Otherwise the type parameter should be a series of one or more 8 character sections. The first four characters are the file type, the second four are the file creator. You can also use the ? character if you do not care what a character is. Here are some useful file types:</p> <pre>TEXT???? list all text files APPL???? list all applications ????KASX list all Panorama database files</pre> <p>You can combine more than one specification into a filter, for example TEXT????????KASX to list all text files and Panorama database files.</p> <p>The listfiles(function normally does not list folders. However, if you precede the filter specification with the f (option-F) character the function will list folders as well as files. For example:</p> <pre>fTEXT???? list all text files and folders f????KASXTEXT???? list databases, text files, and folders</pre> <p>If the filter is empty ("") then ALL files and folders will be included.</p> <p>The function returns a carriage return separated text array. Each item contains a single file name.</p>
info("foldersepchar")		Returns the folder separator character for the current platform (: or \).
info("lineseparator")		This function returns the line separator character on the current platform. On Macintosh systems this is a carriage return. On Windows PC systems this is a carriage return followed by a linefeed (CR-LF).
info("systemfolder")	Page 5429	This function returns a pathid that unambiguously describes the location of the system folder. This pathid can be used in other functions and statements.
info("panoramafolder")	Page 5405	This function returns a pathid that unambiguously describes the location of the folder containing the Panorama application. This pathid can be used in other functions and statements.
filedate(folder,file)		Returns the creation date of the specified file.
fileexists(folder,file)		Checks to see if a file exists. Returns true or false.
fileexists(path)		Checks to see if a file exists. Returns true or false.
fileextension(text)		This function extracts the extension (if any) from a complete path and filename. For example the function fileextension("Alaska:Denali:Image45.jpg") would return .jpg .
filename(text)		This function extracts the filename from a complete path and filename. For example the function filename("Alaska:Denali:Image45.jpg") would return Image45.jpg .
filepath(text)		This function extracts the path from a combined path and filename. For example the function filepath("Alaska:Denali:Image45.jpg") would return the text "Alaska:Denali:" .
filetime(folder,file)		Returns the creation time of the current file.

Function	Reference Page	Description
filetypecreator(folder,file)		Returns 8 characters - 4 characters with the file type and 4 characters with the creator code (for example <code>TEXTTR*ch</code> for BBEdit text files).
folderexists(folder,foldername)		Checks to see if a folder exists. Returns true or false.
foldersepchar()		Returns the folder separator character for the current platform (: or \).
foldercontains(path,file)		Checks to see if a file exists inside the specified folder. Returns true or false.
foldercontents(path)		This function returns a list of all of the contents in a folder (both folders and files).
foldercount(path)		This function returns the total number of files within a folder.
foldersize(path)		This function returns the total size of all of the files within a folder.
fullpath(path)		This function converts a filename or relative path (starting with the : or / symbol) into a full path, including the disk name.
listpatnoramafiles(folder)		Returns a list of Panorama database files in the specified folder (carriage return separated).
listtextfiles(folder)		Returns a list of text files in the specified folder (carriage return separated).
panoramafolder()		This function returns a folderid that unambiguously describes the location of the folder containing the Panorama application. This folderid can be used in other functions and statements.
posixpath(path)		This function converts a MacOS format path and filename into a POSIX path that can be used as a parameter to a shell command. The input parameter must be a FULL path, including the disk name. Relative pathnames are not allowed.
panoramafolder(subpath)		This function makes it easy to reference any subfolder of the Panorama folder. Or you can leave it blank ("") to reference the main Panorama folder.
pathcontains(path,file)		Checks to see if a file exists inside the specified folder. Returns true or false.
pathcontents(path)		This function returns a list of all of the contents in a folder (both folders and files).
pathexists(path)		Checks to see if a path exists. Returns true or false.
posixpath(path)		This function converts a MacOS format path and filename into a POSIX path that can be used as a parameter to a shell command. The input parameter must be a FULL path, including the disk name. Relative pathnames are not allowed.
subfolder(folder,subfolder)		This function returns the folder ID of a subfolder of a specified folder. For example the function <code>subfolder(info("systemfolder"),"Extensions")</code> returns the folder ID of the Extensions folder within the System folder.
subpath(folder,subfolder)		This function returns the path of a subfolder of a specified folder. For example the function <code>subpath(info("systemfolder"),"Extensions")</code> returns the path of the Extensions folder within the System folder.
urlfilename(url)		This function extracts the filename from a complete url.
urlpath(url)		This function extracts the path from a url.

Resource Files

The Macintosh has a special kind of file, called a resource file, for storing multiple chunks of information in a single file. Each chunk of information is called a resource. Each resource may be anything from a single character to tens of thousands of bytes of information. (By the way, Panorama has a special facility to allow resource files to be used even on Windows systems. On Windows these files have the extension .RSR).

Each resource is identified by its **type** and **ID**. The **type** is a four letter designation that identifies what type of data is stored in that resource. There are hundreds of different types of resources, with more new types being created all the time. However, the most common types were defined by Apple in 1984 and are still in use today. This table describes some of the most common types.

Type	Description
MENU	List of the items in a menu
DLOG	Description of a dialog
DITL	Description of the items within a dialog
STR	Single item of text
STR#	Multiple items of text
PICT	Picture
ICON	Icon
ICN#	Multiple icons
cicn	Color icon
CURS	Cursor (mouse pointer)

The resource **ID** is simply a number between 0 and 65535.

Just as a file is identified by its folder and file name, a resource is identified by its type and ID. For example, you may refer to a resource as **MENU 97** or **ICON 2544**.

In addition to a type and ID, a resource may also have a name. However, the name is completely optional. If a resource does have a name, you can identify the resource by its type and name as well as by its type and ID. For example you may refer to a resource as **ICON 2544** or as **ICON Empty Trash Can**.

Before the data in a resource file can be accessed the file must be opened with the `openresource` statement (see “[OPENRESOURCE](#)” on page 5580 of the *Panorama Reference*). To learn more about how to create, modify and use resources see “[Working with Resources](#)” on page 433. The functions that allow you to work with resources are described in the table below.

Function	Reference Page	Description
resourcetypes()	Page 5670	<p>This function creates a text array containing a list of the resource types in all currently open resource files. The result of this function is a carriage return delimited text array. Each element in the array contains a resource type. Each resource type is a four letter text item, for example "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).</p> <p>You can use this function to check if a particular resource type exists, or you can use the function with a pop-up menu or List SuperObject™ to allow the user to select a type of resource for any reason. The formula below will create a text array with resource types.</p> <pre>resourcetypes()</pre> <p>The result of this formula will be a list of resource types something like this:</p> <pre>CNTL CURS INIT KCHR LDEF MACA TPLT SIZE dctb TEXT STR# PICT PAT# MENU</pre> <p>As you can see, the resource types are not listed in any particular order.</p>
resources(type)	Page 5669	<p>This function creates a text array containing a list of resources of a particular type. Type is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu).</p> <p>This function returns a text array containing a carriage return delimited list of all the resources of the specified type. Each element of this list is itself a tab delimited array. The first item is the resource item number. The second item is the resource name (if any).</p> <p>This example builds a list of the TEXT resources in the currently open resource files. (The currently open resource files include Panorama itself and the Macintosh system file, as well as any resource files you have opened with the openresource statement.)</p> <pre>resources("TEXT")</pre> <p>The result will be an array like this.</p> <pre>2001 Error Messages 2002 Command List 2003 Conversion Options</pre>

Function	Reference Page	Description
getresource(type,id)	Page 5311	<p>This function gets a resource from an open resource file and copies it into a variable. Type is the resource type. This must be a four letter text item. Standard resource types include "STR " (Pascal String), "STR#" (multiple strings), "DLOG" (dialog), "DITL" (dialog items), "MENU" (menu). ID is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).</p> <p>The function returns whatever binary data is in the specified resource.</p> <p>This example procedure loads the contents of TEXT resource # 415 into the field LetterBody.</p> <pre>openresource "Letter Templates" LetterBody=getresource("TEXT",415)</pre> <p>All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example loads the contents of the TEXT resource named Thank You #2 into the field LetterBody.</p> <pre>openresource "Letter Templates" LetterBody=getresource("TEXT","Thank You #2")</pre>
getstring(type,id)	Page 5314	<p>This function gets a text resource from an open resource file and copies it into a variable. Type is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type "STR " (Pascal String). (If you specify "" for the type, Panorama will assume "STR ".) ID is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).</p> <p>The function returns whatever text is in the specified resource.</p> <p>This example procedure displays the contents of STR resource # 1296.</p> <pre>openresource "Accounting Messages" message getstring("",1296)</pre> <p>All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example displays the text in the Overflow Error resource.</p> <pre>openresource "Accounting Messages" message getstring("","Overflow Error")</pre>

Function	Reference Page	Description
getnstring(type,id,number)	Page 5307	<p>This function gets a text resource from an open resource file and copies it into a variable. The string is extracted from a STR# resource, which holds a collection of multiple strings in each resource. Type is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type "STR#" (multiple Pascal Strings). (If you specify "" for the type, Panorama will assume "STR#".) ID is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item). Number is the number of the string item within the collection. For example, if the collection contains 6 strings they will be numbered 0, 1, 2, 3, 4, and 5.</p> <p>This function returns whatever text is in the specified item within the specified resource collection.</p> <p>This example procedure displays the contents of STR# resource # 693 item 12.</p> <pre>openresource "Accounting Messages" message getnstring(" ",1296,11)</pre> <p>All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example displays the 12th item in the Errors collection.</p> <pre>openresource "Accounting Messages" message getnstring(" ","Errors",11)</pre>
getstringmatch(type,id,text)	Page 5315	<p>This function searches through a collection of multiple strings in a STR# resource. If it finds a match with the text you supply, it returns the number of the text item within the collection. Type is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type "STR#" (multiple Pascal Strings). (If you specify "" for the type, Panorama will assume "STR#".) ID is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item). Text is the text you want to search for. For a match, this text must be exactly the same as one of the text items in the STR# collection.</p> <p>This function returns a number. If the text does not match any of the text items in the STR# collection, the function will return 0. If there is a match, the function will return the number of the item that matched, starting with 1 for the first item. (Notice that this numbering system is different than the getnstring(function, which starts with 0 for the first item.)</p>

Import/Export Functions

These functions may be used to customize the import and export of data.

Function	Reference Page	Description
import()	Page 5352	<p>This function returns a line of imported data. This function only works in conjunction with the <code>importusing</code> (see “IMPORTUSING” on page 5355 of the <i>Panorama Reference</i>) and <code>arrayfilter</code> statements (see “ARRAY-FILTER” on page 5045 of the <i>Panorama Reference</i>).</p> <p>The <code>import()</code> function returns a line of text. When it is used with the <code>importusing</code> statement, the <code>import()</code> function returns the contents of the line that is currently being imported. Using this function you can process and re-arrange the data as it is being imported.</p> <p>When it is used with the <code>arrayfilter</code> statement, the <code>import()</code> function returns the individual array element currently being processed. Using this function you can process the data in each array element.</p> <p>When it is used at any other time, the <code>import()</code> function returns empty text.</p>
importcell(colNumber)	Page 5353	<p>This function returns one cell of imported data. This function only works in conjunction with the <code>importusing</code> statement (see “IMPORTUSING” on page 5355 of the <i>Panorama Reference</i>). <code>ColNumber</code> is the column of data from the imported text that you want to return. The text being imported is separated into columns by either tabs or commas. The first column is column 0, the next is column 1, etc.</p> <p>The <code>importcell()</code> function always returns text. When it is used with the <code>importusing</code> statement, the <code>importcell()</code> function returns the contents of the specified column from the line that is currently being imported. If the text being imported is comma delimited, the <code>importcell()</code> function will strip off any quotes around the data before returning it. Using this function you can process and re-arrange the data as it is being imported.</p> <p>When it is used at any other time, the <code>importcell()</code> function returns empty text. It will also return empty text if you specify a column number that does not exist in the text being imported.</p>
exportline()	Page 5210	<p>This function generates a tab delimited line of data containing all the fields in the current record. This function is designed to be used with the <code>export</code> (see “EXPORT” on page 5207 of the <i>Panorama Reference</i>) and <code>arraybuild</code> (see “ARRAYBUILD” on page 5038 of the <i>Panorama Reference</i>) statements, but may actually be used anywhere.</p> <p>The function returns a a tab delimited line of data containing all the fields in the current record. Any non-text fields (numeric, date) will be converted to text as they are placed into the tab delimited line. The tab delimited line does NOT include a carriage return on the end.</p>
exportcell(field)	Page 5209	<p>This function takes any database field and converts it to text, using the appropriate pattern if one has been defined in the design sheet. <code>Field</code> is the name of the field to be converted to text.</p> <p>The function always returns a text type data item. The power of the <code>exportcell()</code> function is that it does not require you to know what type of data you are exporting. It simply takes whatever kind of data is in the field (text, number, date, whatever) and converts it into text.</p>

System and Database Information Functions

The functions described in this section allow a formula to access information about the computer system and about the current database.

System Information

These functions return information about the computer system — the keyboard, mouse, memory, clipboard and Panorama itself.

Function	Reference Page	Description
callingdatabase()		This function returns the name of the database that contained the procedure that called this procedure as a subroutine (if any). If a procedure was called as a subroutine by another procedure this function will return the name of the database that contains the calling procedure.
callingprocedure()		This function returns the name of the procedure that called this procedure as a subroutine (if any). If a procedure was called as a subroutine by another procedure this function will return the name of the calling procedure.
callwithindatabase()		This function returns true if the current procedure was called by another procedure in the same database, false if it was called by a procedure in another database or called as a standalone procedure (for example with a button or the Action menu).
clipboard()	Page 5107	This function returns whatever text or value is currently on the clipboard.
currentprinter()		This function returns the name of the current printer (OS X only).
growlrunning()		This function checks to see if Growl is running, and returns true if it is (otherwise false). (Mac OS X only).
homefolder(subpath)		This function returns the folder ID of a subfolder of the current users Home folder. (Mac OS X only).
homepath(subpath)		This function returns the path of a subfolder of the current users Home folder. (Mac OS X only).
info("abort")	Page 5357	The info("abort") function returns true if the user has pressed Command-Period (Macintosh) or Control-Period (Windows). You should use this procedure if you have used the disableabort statement and want to check for Command/Control-Period yourself.
info("availablescreenrectangle")		This function returns a rectangle defining the edges of the main screen (the screen that contains the menu bar). Any area of the screen that is reserved for the operating system (for example the dock in OS X) is excluded from the rectangle. The rectangle is in screen relative coordinates
info("click")	Page 5364	This function returns the location of the last mouse click in screen relative co-ordinates.
info("computername")		This function returns the short name of the computer (OS X only). This is the computer name that needs to be used in terminal sessions.
info("dialogtrigger")	Page 5370	This function returns the name of the last button pressed in a dialog. (Note: This function does not work with standard system dialogs like the Open and Save As dialogs.)
info("error")	Page 5373	This function can be used after an if error statement. It returns the error message that would have been displayed if the error had not been trapped by if error.
info("files")	Page 5376	This function builds a carriage return separated text array containing a list of all the currently open database files.


Function	Reference Page	Description
info("fonts")		This function returns a carriage return delimited array of all available fonts. This list is created when Panorama launches, and is not updated to include any new fonts that may have been added to the system since then.
info("freememory")	Page 5382	This function calculates how much free database memory is available (this does not include scratch memory).
info("keyboard")	Page 5385	This function returns the last key that was pressed.
info("keycode")	Page 5386	This function returns a special numeric code that represents last key that was pressed. This code is unique for every key in the keyboard. For example, the info("keycode") will return a different value for the 1 key on the numeric keypad and the 1 key above the Q key. See the reference page for a complete list of the numeric codes.
info("modifiers")	Page 5397	<p>This function returns the status of the modifier keys. The five different possible modifier keys are:</p> <pre>"shift" "capslock" "option" (returned when ALT key pressed on PC) "command" (returned when Control key pressed on PC) "control" (returned when Right Mouse button clicked on PC)</pre> <p>The info("modifiers") function also returns the status of the last mouse click. If the last mouse click was a double click, info("modifiers") will return "doubleclick". If the last mouse click was a triple click, info("modifiers") will return "tripleclick".</p> <p>If more than one modifier key is active the function will return all of them strung together like this:</p> <pre>"shift option"</pre> <p>You should check for a specific modifier with the <code>contains</code> operator. This example opens the <code>Status</code> form if the user double clicks on a button.</p> <pre>if info("modifiers") contains "double" openform "Status" endif</pre>
info("mouse")	Page 5398	This function returns the current location of the mouse in screen relative co-ordinates.
info("mousedown")	Page 5399	This function returns true or false depending on whether or not the mouse is currently down.
info("mousetilldown")	Page 5400	This function returns true or false depending on whether or not the mouse is currently down and has not been let up since the button was pressed.
info("panoramafolder")	Page 5405	This function returns the folder id (see " Disk Files and Folders " on page 165) of the folder containing the currently running copy of Panorama. This folder id can be used in other functions and statements.
info("panoramatoolsfolder")	Page 5407	This function returns the folder id (see " Disk Files and Folders " on page 165) of the System:Prefs:Panorama Tools folder, if any. If there is no such folder (for example on PC systems) it returns the folder that contains Panorama itself.
info("panoramaname")	Page 5406	This function returns the name of the currently running copy of Panorama. In other words, if you have renamed your copy of Panorama this function will tell what the name is.

Function	Reference Page	Description
info("parameters")		This function returns the number of parameters passed to a procedure.
info("runninghandler")		This function returns true if the current procedure is running as a “handler” procedure. See “ Event Handler Procedures ” on page 394 for more information on this mode.
info("scratchmemory")	Page 5413	This obsolete function returned the amount of memory allocated for scratch memory on OS 9 systems. On OS X and PC systems it always returns 1000000 (one million).
info("screenrectangle")	Page 5414	This function returns a rectangle defining the edges of the main screen (the screen that contains the menu bar). The rectangle is in screen relative coordinates.
info("systemfolder")	Page 5429	This function returns folder id (see “ Disk Files and Folders ” on page 165) of the system folder (Mac OS). This folder id can be used in other functions and statements.
info("trigger")	Page 5433	<p>This function returns information about how the current procedure was triggered. If the procedure was triggered by data entry this function will return the word Key followed by a period and then the key that actually triggered the procedure:</p> <pre>Key.return Key.enter Key.tab</pre> <p>If the procedure was triggered by a button, the function will return the word Button followed by a period and then the name of the button, for example:</p> <pre>Button.Save Button.Calculate Tax Button.Show Chart</pre> <p>If the procedure was triggered by a custom menu, the function will return the word Menu followed by a period, the name of the menu, another period, and then the menu item. Here are some examples:</p> <pre>Menu.Accounting.Aging Menu.Letter.New</pre>
info("version")	Page 5440	<p>This function returns the version of the currently running copy of Panorama. The version number is returned as text, for example 3.5.1. If you want to extract only a portion of the version number you can use the array or array functions. This example will extract the first two numbers of the version. The result will be something like 3.5 or 4.0.</p> <pre>arrayrange(info("version"),1,2,".")</pre>
info("volumes")	Page 5441	This function returns a carriage return delimited array containing the name of each volume (disk) currently mounted on this computer.
info("unixusername")		This function returns the short user name the user has logged in under (Mac OS X only). This is the user name that needs to be used in terminal sessions.
listprinters()		This function returns a carriage return delimited list of the printers available on the system (Mac OS X only).
os9()		This function returns true if running on Mac OS 9, otherwise false.
oswindows()		This function returns true if running on Microsoft Windows, otherwise false.

Function	Reference Page	Description
osx()		This function returns true if running on Mac OS X, otherwise false.
panoramasubpath(child)		This function makes it easy to reference any subpath of the Panorama folder. Or you can leave it blank ("") to reference the main Panorama folder.
parameter(number)	Page 5592	<p>This function is used to transfer data between a main procedure and a subroutine. The main procedure can set up one or more data item parameters as part of the call statement (see "CALL" on page 5086 of the <i>Panorama Reference</i>). The subroutine can retrieve and use these data items using the parameter(function. Number is a number specifying what parameter you want to retrieve. All parameters are numbered, starting with 1 (1, 2,3, 4, etc.). The function returns a data item. This data item may be text or numeric, depending on what kind of data is passed to the sub-routin</p> <p>New in Panorama 5.5: The parameter(function now works with negative parameter numbers. When a negative parameter # is passed it returns information about the parameter data type. This can be used in conjunction with the setparameter statement to modify the data type of the data being passed back by setparameter. Possible data types are: 0 = parameter cannot be modified (might not exist), so has no data type, 1 = text, 2 = compressed text, 3 = compressed text, 4 = picture, 5 = date, 6 = floating point, 7 = integer,8 = fixed 1 digit, 9 = fixed 2 digits, 10 = fixed 3 digits, 11 = fixed 4 digits, -1 = variable (has no data type).</p>

User Information

These functions return information about the person that is currently using the computer. The last three functions can only return useful information if the person has logged in. To learn more about setting up users and database security see the Panorama Database Security Supplement, which is available separately from Pro-VUE for a small charge.

Function	Reference Page	Description
info("user")	Page 5436	<p>This function returns the name of the user of this computer. On Mac OS 9 computers this is the Owner Name which is set with the File Sharing control panel. On OS X this is the name of the logged in user.</p>  <p>On Windows computers this function always returns an empty string ("").</p>
info("userid")	Page 5437	<p>This function works with the Panorama security system. It returns the id (usually initials or the first name) the user has logged in under. If the user has not logged in, this function will return empty text (""). For example, the formula below could be used in a report header (in an auto-wrap text object or Text Display SuperObject™ to show who printed the report and when.</p> <pre>"Printed by: "+info("userid")+ " @"+timepattern(now(),"hh:mm am/pm")</pre>
info("userlevel")	Page 5438	<p>This function works with the Panorama security system. It returns the current user level for this user, a number from 0 to 255. If the user has not logged in, this function will return 0. The example procedure below only allows users with access levels of 25 or higher to use the rest of the procedure.</p> <pre>if info("userlevel")<25 message "Sorry, your access level "+ "does not allow this operation" stop endif (rest of procedure) ...</pre>
info("username")	Page 5439	<p>This function works with the Panorama security system. It returns the name the user has logged in under. If the user has not logged in, this function will return empty text (""). The formula below could be used in a report header (in an auto-wrap text object or Text Display SuperObject™ to show who printed the report and when.</p> <pre>"Printed by: "+info("username")+ " @"+timepattern(now(),"hh:mm am/pm")</pre>
info("unixusername")		<p>This function returns the short user name the user has logged in under (Mac OS X only). This is the user name that needs to be used in terminal sessions.</p>

Variable Information

These functions allow a formula to determine what variables are currently available and to access variables in other databases.

Function	Reference Page	Description
grabfilevariable(file,variable)	Page 5328	<p>This function makes it possible to access a fileglobal (see "FILEGLOBAL" on page 5227 of the <i>Panorama Reference</i>) or permanent (see "PERMANENT" on page 5602 of the <i>Panorama Reference</i>) variable from other databases. (Usually these variables can only be accessed from the database in which they were created.) File is the name of the database that contains the fileglobal or permanent variable. Note: This database must currently be open! Variable is the name of the variable you want to access. In general, this variable name must be enclosed in quotes (unless you are using a formula to calculate the name).</p> <p>The result of this function is whatever value that is contained in the specified variable. This may be text or numeric.</p>
grabwindowvariable(window,variable)	Page 5329	<p>This function makes it possible to access a windowglobal variable (see "WINDOWGLOBAL" on page 5895 of the <i>Panorama Reference</i>) in a different window from the one in which it was created. Window is the name of the window in which the variable was created. (Of course the window must be open!) Variable is the name of the variable you want to access. In general, this variable name must be enclosed in quotes (unless you are using a formula to calculate the name).</p> <p>The result of this function is whatever value that is contained in the specified variable. This may be text or numeric.</p>
info("filevariables")	Page 5377	This function builds a carriage return separated text array containing a list of the currently allocated fileglobal variables in the current database.
info("globalvariables")	Page 5383	This function builds a carriage return separated text array containing a list of the currently allocated global variables.
info("localvariables")	Page 5389	This function builds a carriage return separated text array containing a list of the currently allocated local variables.
info("windowvariables")	Page 5449	This function builds a carriage return separated text array containing a list of the currently allocated window variables for the current window.

Database Information

These functions return information about the currently active database.

Function	Reference Page	Description
constantvalue(fieldorvariable)		This function converts a field or variable into an equivalent constant value. If the field or variable contains text the result will be quoted. The result of this function can be used in an execute statement.
countsummaries(level)		This function counts the number of summary records in the current database. The level parameter should be from 0 to 7. If 0, all summary records will be counted. If 1 to 7 then only that specific level will be counted.

Function	Reference Page	Description
datatype(fieldvariablename)	Page 5142	<p>This function determines what kind of data is in a field or variable: text, number, etc. Fieldvariablename is the name of the field or variable that you want to get information about. To get information about a variable the variable name must be enclosed in quotes. The quotes are optional when accessing information about a field. The function returns the type of data from the list below:</p> <ul style="list-style-type: none"> Text Compressed (Choice) Picture Date Floating Point Integer Fixed 1 Digit (#.#) Fixed 2 Digits (#.##) Fixed 3 Digits (#.###) Fixed 4 Digits (#.####) <p>Note: The Compressed, Picture, and Date types can only occur if the datatype(function is used with a field as the parameter. Variables cannot contain data of these types (for a date, the data type is Integer).</p>
datavalue(fieldorvariable)		This function returns the value of a field or variable. The advantage of this function is that you can calculate the name of the field or variable at the time the formula is evaluated.
dbcheckopen(database)		This function returns true if the specified database is currently open, otherwise false.
dbfolder()		This function returns the folder ID of the folder the current database is located in.

Function	Reference Page	Description
dbinfo(option,database,)	Page 5150	<p>This function gets information about a database: what forms it contains, what fields, what flash art pictures, etc. There are two parameters: option and database.</p> <p>Database is the name of the database you want to get information about. This must be a database that is currently open. If you want to get information about the current database you can use the info("databasename") function or simply use empty text ("").</p> <p>Option controls what kind of information this function will retrieve. There are about a half dozen possible options: "fields", "forms", "procedures", "crosstabs", "flash art", "folder", "level" and "autosave". The "fields" option produces a text array (with carriage return separators) containing a list of the fields in the database. (If a field name contains carriage returns, they are converted to spaces before being placed into the array.) The "forms" option produces a text array (with carriage return separators) containing a list of the forms in the database. The "procedures" option produces a text array (with carriage return separators) containing a list of the procedures in the database. The "crosstabs" option produces a text array (with carriage return separators) containing a list of the crosstabs in the database. The "flash art" option produces a text array (with carriage return separators) containing a list of the flash art in the database's Flash Art™ gallery. The "folder" option produces a folder id for the folder containing the database. (See "Disk Files and Folders" on page 165.) The "level" option returns a number that indicates the privilege level for this database: 0 = author mode, 1 = user mode, or 2 = custom mode. The "autosave" option returns the number of minutes between automatic saves, or zero if the auto-save option is turned off. (See also "SETAUTOSAVE" on page 5739 of the <i>Panorama Reference</i>.) The "fileglobals" option lists all fileglobal variables associated with the database specified by the second parameter, as does the "filevariables" option. The "fieldtypes" option returns binary data with one byte per field. Each byte indicates the field type of the corresponding field, and will be one of the following values: 0=text, 4=date, 5=float, 6=integer, 7-10=fixed point. The "records" option returns the number of records in the database. The "selected" option returns the number of records in the database. The "changes" option returns the number of changes since last save.</p> <p>This example uses dbinfo(to calculate the number of forms in the current database.</p> <pre>arraysize(dbinfo("forms",""),¶)</pre>
dbmetatag(database,attributes)		<p>This function returns a piece of metadata information for the specified database (if any). This second parameter specifies the metadata item you want to retrieve - legal items are: Version, Title, Project, Authors, Keywords, Description. These are all case sensitive. For example, you can retrieve the document keywords with the formula <code>dbmetatag(myDatabase,"Keywords")</code>.</p>
dbmetatagclose()		<p>This function returns the tag that appears after the metadata database information. This tag helps the Tiger (Mac OS X 10.4) metadata import function to locate the metadata.</p>
dbmetatagdictionary(database)		<p>This function returns the metadata database information for the specified database (if any). This metadata contains information like the database author, keyword and description.</p>
dbmetatagopen()		<p>This function returns the tag that appears in front of the metadata database information. This tag helps the Tiger (Mac OS X 10.4) metadata import function to locate the metadata.</p>
dbname()		<p>This function returns the name of the current database.</p>

Function	Reference Page	Description
dbpath()		This function returns the path of the folder the current database is located in.
dbsubfolder(subpath)		This function returns the folder ID of a subfolder within the same folder as the current database. For example if the current database is in the folder "MyDrive:My Stuff", the function <code>dbsubfolder("Images")</code> returns the folder ID of the "MyDrive:My Stuff:Images" folder.
emptydatabase()		This function returns true if the current database is empty, false if it is not.
emptyline()		This function returns true if the current line (all fields) is empty, false if it is not.
fieldmax(fieldname)	Page 5219	This function returns the maximum number of characters that can be stored in a field. If this is not an SQL client database, this number is always 65535. If this is an SQL client, this function returns the length of the corresponding SQL field in the server database.
fieldstyle(fieldname)	Page 5221	<p>This function determines the style and color of a field (see "Data Style and Color" on page 474). Fieldname is the name of the field that you want to determine the style of. The function returns a text data item listing all the styles that apply to this field in the current record. The possible styles are:</p> <ul style="list-style-type: none"> bold italic underline shadow black red green blue cyan magenta yellow <p>The example below selects all the records where the name is bold.</p> <pre>select fieldstyle(Name)="bold"</pre> <p>It there is more than one style for a cell, this function will list each one. The example below will select all records where the name is italic, even if other styles also apply (for example bold italic or underline italic).</p> <pre>select fieldstyle(Name) contains "italic"</pre> <p>This final example selects all the records where the name is plain (no styles at all).</p> <pre>select fieldstyle(Name)=" "</pre>
getautonumber()		This function returns the automatically generated number for the next record that will be added to the database.
getproceduretext(database, procedure)		This function returns the contents (source) of a procedure. The first parameter is the name of the database (or "" for the current database) and the second parameter is the name of the procedure.
grabfieldtype(database,field)		This function gets the type of a database field in any open database. The result is an integer: 0=text, 4=date, 5=floating point, 6-10=integer.

Function	Reference Page	Description
listchoices(field,separator)	Page 5475	<p>This function builds a text array containing a list of all the values stored in a specified field. (Note: this function is not related to the choices data type.) There are two parameters: field and separator. Field is the name of the field that contains the values you want to build a list of. Separator is the separator character for the text array you are building (see “Text Arrays” on page 93).</p> <p>The listchoices(function scans the specified field and builds a list of all the values stored in that field. The list is returned in the format of a text array. Here is a formula that builds a list of the states in the current database.</p> <pre>listchoices(State,¶)</pre>
seq()	Page 5727	<p>This function returns a sequential numbers (1, 2, 3, etc.). This function only works in conjunction with the formulafill, select, find and arrayfilter statements.</p> <p>When it is used with the formulafill, find or select statements (see “FORMULAFILL” on page 5274, “FIND” on page 5245, and “SELECT” on page 5710 of the <i>Panorama Reference</i>), the seq() function return a sequential number for each record (the first selected record is 1, the second is 2, etc.).</p> <p>When it is used with the arrayfilter statement (see “ARRAYFILTER” on page 5045 of the <i>Panorama Reference</i>), the seq() function returns a sequential number for each element in the array being processed (the first array element is 1, the second is 2, the third is 3, etc.).</p> <p>When it is used at any other time, the seq() function returns the number 1.</p> <p>This procedure uses the seq() function to select the first 10 records in the database:</p> <pre>select seq()≤10</pre>

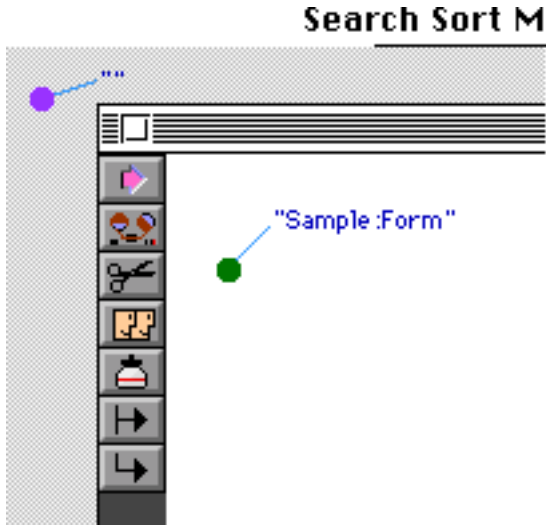
Function	Reference Page	Description
uniqueid(field,root)	Page 5870	<p>This function is designed for generating unique ID codes for each record in a database. The function generates ID codes with a text root and a numeric suffix (for example Jeff261). By using the machine name as the text root you can guarantee that the ID will be unique even for multiple copies of the database on different machines.</p> <p>The function has one parameters: field and root. Field is the name of the field that will contain the ID code. The function needs to know the name of this field so that it can scan the field to find an ID code that has not been used yet. The field name should be surrounded by quotes. For example, if the name of the field is ID, you should use "ID" as the parameter. Root is the text root that the ID code will be based on. This root may contain any kind of character, but it should not end with a numeric digit. To get a root that will be unique for each different computer you have, use the info("user") function for the root. This function returns the user name specified in the Sharing Setup control panel.</p> <p>Although you may find other uses for it, the uniqueid(function was designed specifically for creating unique Smart Merge serial numbers. Whenever a new record is added to a database that supports Smart Merge you must make sure that the ID and Modified fields are filled in. The best way to do this is to add a .NewRecord automatic procedure to your database. The two lines shown below will fill in the proper values.</p> <pre>Modified=superdate(today(), now()) ID=uniqueid("ID",info("user"))</pre> <p>The uniqueid(function will scan the ID field to find the next serial number available. For example, if you are using a computer with a user name of Sam and the highest Sam serial number is 296, the uniqueid(function will return the value Sam297.</p>
info("bof")	Page 5360	This function returns true if the database is currently on the first visible record. (Note: "bof" stands for "beginning of file".)
info("changes")	Page 5363	This function returns the number of changes that have been made to the current database since the last time it was saved.
info("cursorrectangle")	Page 5365	This function returns a rectangle defining the edges of the current data cell (if any). The rectangle is in screen relative coordinates (use the xytoxy((see " Rectangles " on page 149) function to convert to window or form relative co-ordinates).
info("databasename")	Page 5367	This function returns the name of the current database. If the database name has a .pan suffix, that suffix is not included.
info("datatype")	Page 5368	<p>This function returns the data type of the current field. The function returns a number from 0 to 10:</p> <ul style="list-style-type: none"> 0 Text 1 Choice 2 Choice 3 Picture 4 Date 5 Floating Point 6 Integer 7 Fixed 1 Digit (##) 8 Fixed 2 Digits (###) 9 Fixed 3 Digits (####) 10 Fixed 4 Digits (#####)

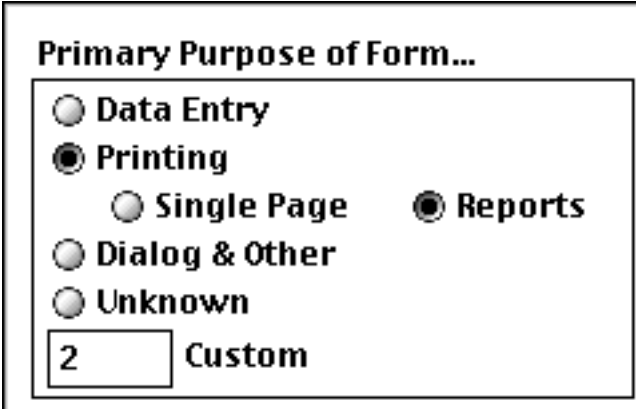
Function	Reference Page	Description
info("empty")	Page 5371	<p>This function returns true or false depending on the result of the last select operation. If no records were selected the function will return true, otherwise it will return false. The procedure below selects all records that are "Ready", whatever that means. If there are any ready records, the procedure prints them.</p> <pre> select Status="Ready" if info("empty") message "Nothing ready today!" stop endif print dialog field Status formulafill "Printed" </pre>
info("eof")	Page 5372	This function returns true if the database is currently on the last visible record. (Note: "eof" stands for "end of file".)
info("expandable")	Page 5374	This function checks to see if the current record is a collapsed summary record. It returns true if the record is a collapsed summary record, false if it is a data record or an already expanded summary record.
info("fieldname")	Page 5375	This function returns the name of the current field. See also the info("modifiedfield") function below.
info("found")	Page 5381	This function returns true or false depending on the result of the last find or next statement (see " FIND " on page 5245 of the <i>Panorama Reference</i>). If the last find or next found something, this function will return true. Otherwise it will return false.
info("modifiedfield")		This function returns the name of the field that was just modified. The most recently modified field is not always the same as the current field (see the info("fieldname") function above). For example if a checkbox or radio button has just been clicked this function will return the name of the field associated with that object rather than the current field. This function is designed to be used in the .ModifyRecord and .ModifyFill procedures, the results may not be valid if the function is used in any other context.
info("proceduredatabase")		<p>This function returns the name of the database that contains the procedure itself, even if another database is currently active. It's primarily useful when the farcall statement is used to call the procedure (or when the procedure is being used as a custom statement). Here's an example that opens the Help Window form in the database that contains the procedure, even if a window from another database is currently on top,</p> <pre> window info("proceduredatabase")+":SECRET" openform "Help Window" </pre>
info("records")	Page 5410	<p>This function returns the total number of records in the current database. To find out the number of selected records, use info("selected") (see below).</p> <p>This example checks to see if all records are selected. If some records are not selected, the procedure does a selectall statement.</p> <pre> if info("selected") <info("records") selectall endif </pre>
info("selected")	Page 5415	This function returns the number of selected records in the current database. To find out the total number of records, use info("records") (see above).

Function	Reference Page	Description
info("stopped")	Page 5426	This function returns true or false depending on the result of the last uprecord , downrecord , left or right statement. If the statement could not move the active cell because the active cell was already as far as it could go, the function will return true. Otherwise it will return false.
info("summary")	Page 5428	This function returns the summary level of the current record, from 0 (data record) to 7 (see " 3-Step Summarizing " on page 365 of the <i>Panorama Handbook</i>).
info("tabdown")	Page 5430	This function returns true if the tab down option is on, false if the tab down option is off (see " Tab Down " on page 277 of the <i>Panorama Handbook</i>).
info("visible")		This function returns true/false result based on whether or not the current record is visible. Useful for situations where Panorama may scan invisible records, for example the arraybuild and select statements, also the Scrolling List SuperObject.

Window, Form and Report Information

These functions return information about the current database and its windows, forms reports and objects.

Function	Reference Page	Description
extrapages(pagelist)	Page 5213	<p>This function is used to control the printing of extra pages. This function must be used in an auto-wrap text object, it has no effect in any other situation. Pagelist is a text item listing the extra pages that should be printed. For example, if you want to print data tiles 3 and 5 the page list should be "35". See "Selectively Printing Multiple Pages per Record" on page 1115 of the <i>Panorama Handbook</i>.</p>
findwindow(point)	Page 5250	<p>This function checks to see if a point (in screen relative co-ordinates) is inside any Panorama window. If it is inside a window, this function returns the name of the window. Point is a point, which must be in screen relative co-ordinates. All measurements are in pixels (1 pixel = 1/72 inch).</p> <p>The function returns a text item. If the point is inside a Panorama window, the function returns the name of the window. You can use the window statement to bring this window to the top. If the point is not inside any Panorama window the function returns empty text ("").</p> <p>This illustration shows a window and two points. The green point is inside the window, so the findwindow(function will return the window name, in this case Sample:Form. The purple point is not inside the window, so the findwindow(function will return "".</p> 

Function	Reference Page	Description
formtype(database,form)	Page 5270	<p>This function returns the form type (a number) for any form in any open database. The form type is a number that you can set up using the Form Comment dialog in the Setup menu (see “Form Comments” on page 732). Database is the name of the database that contains the form. The database must be currently open. If this parameter is empty text (""), the current database is assumed. Form is the name of the form.</p> <p>This function returns a number (integer) from 0 to 255. The value of this number depends on the Primary Purpose of Form area of the Form Comment dialog (in the graphics mode Setup Menu.) There are predefined radio buttons for 1) Data Entry, 2) Printing, and 3) Dialog and other. Or you may enter any value from 0 to 255 in the Custom area. The default value of a form is 0 (unknown).</p> 
listwindows(file)	Page 5478	<p>This function builds a text array containing a list of all the open windows associated with a particular file. File is the name of the database file that you want to list the windows of. This should be the name of an open database. If the file parameter is empty (""), the listwindows(function will list all open windows, no matter what database they are in.</p> <p>The function scans the windows and builds a text array using carriage returns (¶) as separators (see “Text Arrays” on page 93). The windows are listed in order from front to back.</p>

Function	Reference Page	Description
objectinfo(option)	Page 5557	<p>This function returns information about a graphic object: its location, size, color, font, etc. This function must be used in combination with either the object (see “OBJECT” on page 5555 of the <i>Panorama Reference</i>), selectobjects (see “SELECTOBJECTS” on page 5721 of the <i>Panorama Reference</i>) or changeobjects (see “CHANGEOBJECTS” on page 5097 of the <i>Panorama Reference</i>) statement. It can also be used in a formula inside an object on a form — in that case it will always refer to the object currently being drawn (an object could find out its own rectangle, for example).</p> <p>Option is the type of information you want to retrieve about an object. You must pick the option from the list below:</p> <pre> objectinfo("rectangle") objectinfo("ID") objectinfo("name") objectinfo("fieldname") objectinfo("type") objectinfo("custom") objectinfo("font") objectinfo("textsize") objectinfo("textstyle") objectinfo("alignment") objectinfo("color") objectinfo("selected") objectinfo("locked") objectinfo("expandable") objectinfo("expandshrink") objectinfo("tile") objectinfo("text") objectinfo("fillpattern") objectinfo("linepattern") objectinfo("linewidth") objectinfo("count") objectinfo("boundary") </pre> <p>See the reference page for details on each of these options.</p>
overflow()	Page 5587	<p>This function is used with auto-wrap text objects and an overflow report tile to print text or graphics that won't fit on a single page. For example, you can use this function to help print multiple page letters. See “Printing Data that Overflows a Page” on page 1116 of the <i>Panorama Handbook</i> for more information on using this function.</p>

Function	Reference Page	Description
textdisplay(color,style)	Page 5852	<p>This function works with Text Display SuperObjects™. By using this function as the first part of the formula in a Text Display SuperObject™ you can control the color and style of the text on the fly (see “Controlling Text Display Color and Style on the Fly” on page 619). For example, you can automatically display all negative numbers in red. (Advanced note: The textdisplay(function actually generates a special header that is intercepted and removed by the Text Display SuperObject™. The header contains information the Text Display SuperObject™ uses to select the style and color.)</p> <p>This function has two parameters: color and style. Color is the color that should be used to display the text. See the rgb(function. If you pass "" for this parameter the text will be displayed in the normal color for this object. Style is the style or combination styles that should be used to display the text. For a single style by itself simply use the name of the style: "Plain", "Bold", "Italic", "Underline", "Outline" or "Shadow". If you want to combine multiple styles together you must specify the style numerically. Add up the numbers for the styles you want from the table listed below. For example, for bold italic text the style should be 3.</p> <ul style="list-style-type: none"> 0 Plain 1 Bold 2 <i>Italic</i> 4 Underline 8 Outline 16 Shadow
info("activesuperobject")	Page 5358	This function returns the name of the currently active text editor or word processor SuperObject, if any. If no such object is currently being edited, the function returns empty text ("").
info("buttonrectangle")	Page 5361	This function returns a rectangle defining the edges of the button that was clicked on (needless to say, this function should be used in a procedure that is triggered by a button). The rectangle is in screen relative coordinates (use the xytoxy(function to convert to window or form relative co-ordinates).
info("cursorrectangle")	Page 5365	This function returns a rectangle defining the edges of the current data cell (if any). The rectangle is in screen relative coordinates (use the xytoxy((see “ Rectangles ” on page 149) function to convert to window or form relative co-ordinates).
info("formcolor")	Page 5378	This function returns the background color of the current form (see “ Colors ” on page 154). If the current window does not contain a form, the function will return empty text ("").
info("formcomment")	Page 5379	This function returns the form comment that has been set up for the currently open form (if any). See “ Form Comments ” on page 732.
info("formname")	Page 5380	This function returns the name of the current form. If the current window does not contain a form, the function will return empty text ("").

Function	Reference Page	Description																														
<p>info("matrixcell")</p>	<p>Page 5391</p>	<p>This function is designed to be used with Matrix SuperObjects™ (see “Matrix Co-Ordinates (What cell is this?)” on page 953 of the <i>Panorama Handbook</i>). The function returns the cell number within the matrix, starting with 1 in the upper left hand corner. If the matrix order is horizontal, then the cell numbers will be consecutively numbered from left to right in each row. If the matrix order is vertical, then the cell numbers will be consecutively numbered from top to bottom in each column.</p> <p>The illustration shows two matrixes with the cell number displayed in the upper right hand corner. The matrix on the left has vertical cell order, the matrix on the right has horizontal cell order.</p> <div style="display: flex; justify-content: space-around; align-items: center;"> <div style="text-align: center;"> <p>vertical cell order</p> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td>6</td><td>11</td></tr> <tr><td>2</td><td>7</td><td>12</td></tr> <tr><td>3</td><td>8</td><td>13</td></tr> <tr><td>4</td><td>9</td><td>14</td></tr> <tr><td>5</td><td>10</td><td>15</td></tr> </table> </div> <div style="text-align: center;"> <p>horizontal cell order</p> <table border="1" style="border-collapse: collapse; text-align: center;"> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td></tr> <tr><td>7</td><td>8</td><td>9</td></tr> <tr><td>10</td><td>11</td><td>12</td></tr> <tr><td>13</td><td>14</td><td>15</td></tr> </table> </div> </div>	1	6	11	2	7	12	3	8	13	4	9	14	5	10	15	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	6	11																														
2	7	12																														
3	8	13																														
4	9	14																														
5	10	15																														
1	2	3																														
4	5	6																														
7	8	9																														
10	11	12																														
13	14	15																														
<p>info("matrixcelldata")</p>		<p>This function returns the data associated with the current matrix cell. This function is only valid if a formula has been defined for this Super Matrix Object. Note: It's up to you what to do with this data. You can display it using a Text or Flash Art object, or simply ignore it if you want. The Matrix object doesn't display this data automatically.</p> <p>Example: The formula below could be used in a Text Display SuperObject to display the cell number and data.</p> <pre style="text-align: center;">str(info("matrixcell"))+" "+info("matrixcelldata")</pre>																														
<p>info("matrixcolumn")</p>	<p>Page 5392</p>	<p>This function is designed to be used with matrix SuperObjects™ (see “Super Matrix Objects” on page 939 of the <i>Panorama Handbook</i>). The function returns the column number, starting with 1 for the left hand column and increasing by one for each column to the right.</p> <p>The illustration shows the columns and rows for a Matrix SuperObject.</p> <div style="text-align: center;"> <p>--- rows ---</p> <table style="margin: auto;"> <tr> <td></td> <td style="text-align: center;">1</td> <td style="text-align: center;">2</td> <td style="text-align: center;">3</td> </tr> <tr> <td style="text-align: center;">1</td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> </tr> <tr> <td style="text-align: center;">2</td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> </tr> <tr> <td style="text-align: center;">3</td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> </tr> <tr> <td style="text-align: center;">4</td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> </tr> <tr> <td style="text-align: center;">5</td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> <td style="border: 1px solid black; width: 40px; height: 20px;"></td> </tr> </table> <p style="text-align: center;">--- columns ---</p> </div>		1	2	3	1				2				3				4				5									
	1	2	3																													
1																																
2																																
3																																
4																																
5																																
<p>info("matrixdata")</p>		<p>This function returns the all of the data associated with the current matrix. This function is only valid if a formula has been defined for this Super Matrix Object. Note: It's up to you what to do with this data. You can display it using a Text or Flash Art object, or simply ignore it if you want. The Matrix object doesn't display this data automatically.</p> <p>Example: The formula below could be used in a Text Display SuperObject to display the data and the cell number and position, for example Orange [4 of 17] .</p> <pre style="text-align: center;">info("matrixcelldata")+ " [" +str(info("matrixcell"))+" of "+ str(arraysize(info("matrixdata")+ , info("matrixseparator")) +"]"</pre>																														

Function	Reference Page	Description
info("matrixrow")	Page 5394	This function is designed to be used with matrix SuperObjects™. The function returns the row number, starting with 1 for the top and increasing by one for each row as you go down. The illustration above the columns and rows for a matrix SuperObject.
info("matrixseparator")		This function returns the separator associated with the current matrix. This function is only valid if a formula has been defined for this Super Matrix Object. Example: The formula below could be used in a Text Display SuperObject to display the data and the cell number and position, for example Orange [4 of 17]. <pre>info("matrixcelldata")+ " ["+str(info("matrixcell"))+ " of "+str(arraysize(info("matrixdata"))+ , info("matrixseparator"))+"]"</pre>
info("maximumwindow")	Page 5395	This function returns the largest possible rectangle for this window. This can be controlled by setting up an Auto Grow SuperObject for this window (see " Maximum Window Size " on page 929 of the <i>Panorama Handbook</i>).
info("minimumwindow")	Page 5396	This function returns the smallest possible rectangle for this window. This can be controlled by setting up an Auto Grow SuperObject for this window (see " Maximum Window Size " on page 929 of the <i>Panorama Handbook</i>).
info("pagecount")		The new info("pagecount") function calculates the total number of pages that will be printed. For example, if you wanted to print Page 1 of 4 , Page 2 of 4 , etc. on the top of each page you would use a formula of <pre>"Page "+str(info("pagenumber"))+" of " +str(info("pagecount"))</pre> This function is only valid when used in an object in a form (Text Display SuperObject, auto-wrap text object, etc.) that is being printed.
info("pagenumber")	Page 5403	When printing, this function returns the current report page number. This function is designed to be used as part of an auto-wrap text object or Text Display SuperObject™ in a report form. See " Page Numbers " on page 1100 of the <i>Panorama Handbook</i> .
info("reportcolumns")	Page 5411	This function returns the number and direction of report columns that have been set up for the currently open form (if any). The function returns a text string that contains the number of columns followed by the direction, for example " 3 Down " or " 2 Across ". See " Controlling the Number of Columns " on page 1139 of the <i>Panorama Handbook</i> .
info("rulers")	Page 5412	This function returns the current measurement units for the ruler in this form. The function may return five possible values: Inches , Centimeters , Pixels , Deca-Pica , or Deca-Elite . See " Rulers " on page 506 of the <i>Panorama Handbook</i> .
info("screenrectangle")	Page 5414	This function returns a rectangle defining the edges of the main screen (the screen that contains the menu bar). The rectangle is in screen relative coordinates.

Function	Reference Page	Description
<code>info("typeofwindow")</code>	Page 5435	<p>This function determines what type of window the current window is. The window may be one of the types listed below:</p> <ul style="list-style-type: none"> Data Sheet Form (Data Mode) Draw (Graphics Mode) View As List Design Sheet Cross Tab Sheet Floating Input Window Procedure Flash Art Gallery Clipboard Memory Usage Print Preview
<code>info("windowbox")</code>	Page 5442	<p>This function returns the dimensions of the current window in screen relative co-ordinates. All four dimensions are returned in a text string, for example "34 123 490 630". This is a "classic" Panorama function that is retained for compatibility with older databases. For new applications we recommend using the <code>info("windowrectangle")</code> function (see below).</p>
<code>info("windowdepth")</code>	Page 5443	<p>This function returns the pixel depth of the current window. This table shows the possible values:</p> <ul style="list-style-type: none"> 1 Black and White 2 4 color 4 16 color 8 256 color 16 Thousands of colors 32 Millions of colors <p>If the window crosses over two monitors with different pixel depths, the <code>info("windowdepth")</code> function will return the lower value.</p> <p>The formula below could be used in a Flash Art object. If this is a black and white monitor it displays the picture <code>bwSky</code>, otherwise it displays the picture <code>Sky</code>.</p> <pre>?(info("windowdepth")=1, "bwSky", "Sky")</pre>
<code>info("windowname")</code>	Page 5445	This function returns the name of the current window.
<code>info("windowrectangle")</code>	Page 5446	This function returns a rectangle (see " Rectangles " on page 149) defining the edges of the current window. The rectangle is in screen relative coordinates.
<code>info("windows")</code>	Page 5447	This function builds a carriage return separated text array (see " Text Arrays " on page 93) containing a list of all the currently open windows. The windows are listed in order from front to back.

Function	Reference Page	Description
info("windowtype")	Page 5448	<p>This function determines what number type the current window is. The window number may be one of the listed below:</p> <ul style="list-style-type: none"> 2 = Data Sheet 5 = Form (Data Mode) 6 = Draw (Graphics Mode) 15 = View As List 7 = Design Sheet 10 = Cross Tab Sheet 1 = Desk Accessory 3 = Floating Input Window 8 = Procedure 11 = Flash Art Gallery 13 = Clipboard 12 = Memory Usage 14 = Print Preview
updatingwindow()		This function returns true if called from a procedure that is part of displaying an object in a form, otherwise it returns false.

Server Database Information (Panorama Enterprise)

These functions provide information about the server database associated with the current Panorama database (if any).

Function	Reference Page	Description
adjustservervariable(database, variable,adjustvalue)		This function adjusts the value of a server variable. If adjustvalue is a number the server variable must also be numeric, and is incremented or decremented. If adjustvalue is text, it is appended to the server variable.
dbserverdomain()		This function returns the server domain for the database. If this is running on a server, it always returns the servers domain. If running on a client (presumably for testing) it will return the ip address of the server this client is hosted on.
dbshared()		This function returns true if the current database is shared, false if it is not.
dbwebpublish()		This function returns true if the current database is web published, false if it is not.
info("plugandrung")	Page 5408	<p>This function tells how Panorama will resolve conflicts between the client and server when the client database is reconnected to the server after being used off line. There are four possible modes:</p> <p>off This is the value that will be returned if this is a single user Panorama database that is not linked to an SQL server database.</p> <p>client This means that if a record has been modified by both the client and the server, the clients changes will be kept and the servers changes will be discarded.</p> <p>server This means that if a record has been modified by both the client and the server, the servers changes will be kept and the clients changes will be discarded.</p> <p>manual This means that if a record has been modified by both the client and the server, the user will be presented with a list of the changed record and allowed to "cherry pick" which records to keep.</p>
info("serverfile")	Page 5420	This function returns the name of the SQL database linked to this Panorama database, if any.

Function	Reference Page	Description
info("serverrecordid")	Page 5421	This function returns the internal serial number on the server of the current Panorama record. This number is guaranteed to be unique in this database if this is a SQL connected database. If this is a standalone database, this function will return zero for all records. This function will also return zero for new records created while disconnected to the server. These new records are not assigned an internal serial number until the next time the database is synchronized with the server.
info("serverrecordts")	Page 5422	This function returns the internal "time stamp" number Panorama uses to determine which records need to be synchronized. By itself, this number is basically meaningless. However, if the time stamp for record A is higher than record B, then record A was edited later than record B. If this is a standalone database, this function will return zero for all records. This function will also return zero for new records created while disconnected to the server. These new records are not assigned an internal time stamp until the next time the database is synchronized with the server. This function is intended for debugging purposes only. It is included here for completeness. However, it could possibly have useful non-debugging purposes.
info("serverstatus")	Page 5423	This function returns the connection status of the SQL database linked to this Panorama database, if any. There are four possible values: Standalone (Read/Write) This is the value that will be returned if this is a single user Panorama database that is not linked to an SQL server database. Connected (Read/Write) This is the value that will be returned if this is a multi user Panorama database that is linked to an SQL server database, and the link is currently open with full record locking. No Connection (Read/Only) This is the value that will be returned if this is a multi user Panorama database but there is currently no network connection to the SQL server database. For example the user might be using this database on a laptop computer with no connection to the server. This database does not allow off-line editing, so the database cannot be edited. Standalone (Read/Write) This is the value that will be returned if this is a multi user Panorama database but there is currently no network connection to the SQL server database. For example the user might be using this database on a laptop computer with no connection to the server. This database does allow off-line editing, so the database may be edited. The changes made off-line will be saved and synchronized the next time the server is available.
info("servertimeout")	Page 5424	This function returns the maximum time Panorama will keep a record locked with no keyboard or mouse activity. This timeout can help prevent a user from starting to edit a record and then walking away from the computer and leaving the record locked and unavailable to other users indefinitely. The time interval is specified in seconds. A timeout value of zero indicates no timeout (infinite time).
info("subsetformula")	Page 5427	The info("subsetformula") function returns the formula used to extract the current local subset from the server database. If the local database contains a copy of the entire server database (select all) the result will be an empty string ("").
serverdomain()		This function returns the domain name or IP address of a server. The server must be currently available.
serverrunning()		This function returns TRUE if this copy of Panorama is running as a server, or false if it is running in single user or as a client.
servervariable(database,variable)		This function gets the value of a server variable (a permanent variable on the server). The server variable must already be set up with the SetServer-Variable statement.

Function	Reference Page	Description
<code>serverdatabasename(database)</code>		This function returns the name of the server associated with a shared database. If the database name is "" the server name for the current database will be returned.
<code>servername(database)</code>		This function returns the server database name associated with a shared database. This is the name of that database on the server (which may be different from the name on a client). If the database name is "" the server database name for the current database will be returned.
<code>sharedusers(database)</code>		This function returns a list of users that are currently sharing a database. The specified database must currently be connected to the server on this computer. If the database name is left blank then the current database is assumed. If the database name is * then all users on the server will be listed (the current database must be a shared database). If the database is not connected, or is not a sharable database, the result will be "". If this is a connected database the result will be a carriage return separated array, with each line containing the session id, user name, and user's computer's name separated by tabs.

Custom Functions

Panorama doesn't limit you to the built-in functions that are supplied with a Panorama. In fact, you can actually create your own user defined functions that can be used in any formula. To build a custom function you assemble it from the functions and operators that already exist. For example, you could define a new `money()` function with one parameter:

```
money(number)
```

This custom function can be defined using Panorama's built in `pattern()` function, like this.

```
pattern(number, "#, .##")
```

Once the function has been defined you can use it in any formula. For example, the formula

```
money(54321.5678)
```

would result in the text value **54,321.56**.

The Custom Functions Wizard

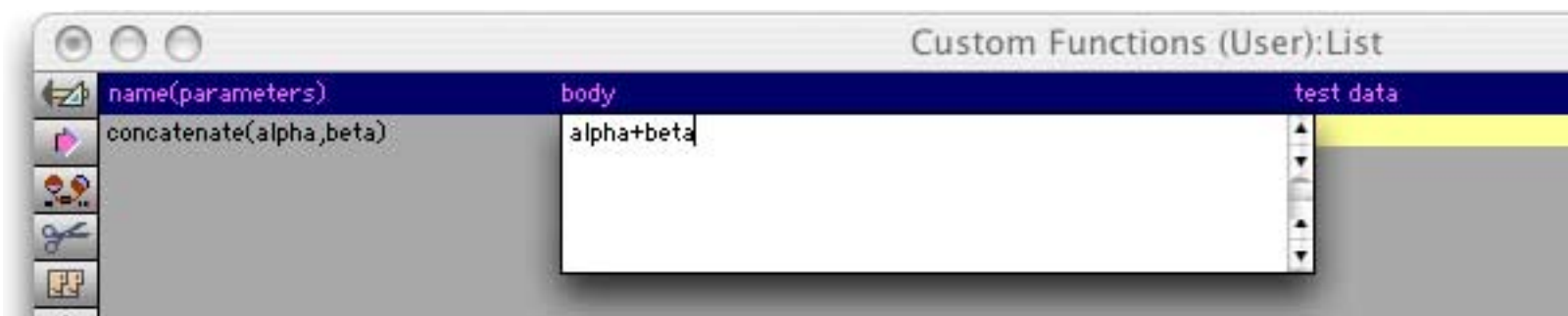
To create your own custom functions you'll use the **Custom Functions (User)** wizard. You'll find this wizard in the Developer Tools submenu of the Wizard menu. This wizard has a view as list form with four columns (three of which are shown in the illustration below. Each line is a slot for a custom function.



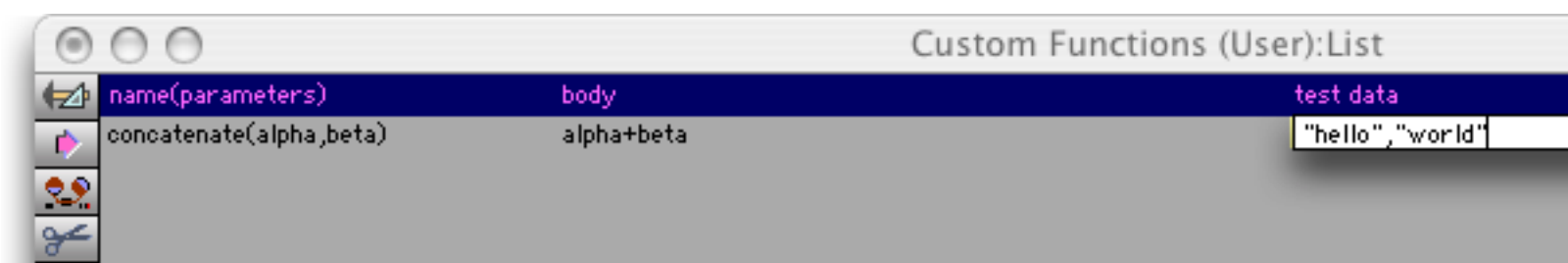
To create a new custom function, use the **Add New Record** tool to add a line to the database. In the leftmost column, type in the name of the function and parameters, like this:



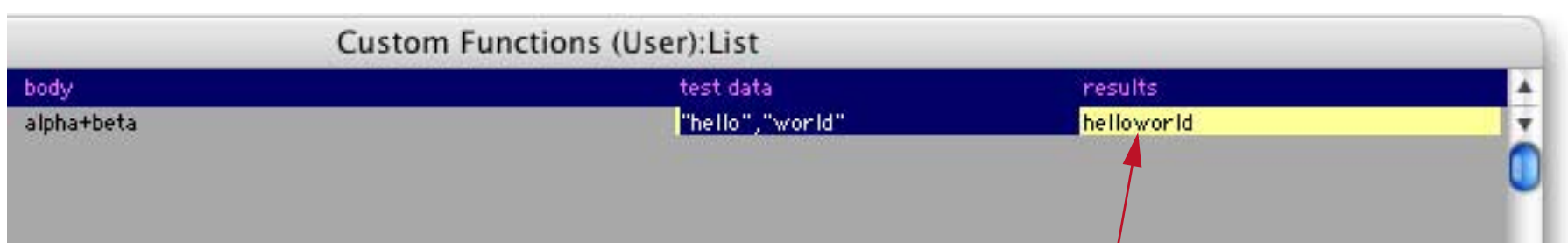
Now tab over to the next column (labeled "body"). Enter the formula that should be used to calculate the custom function's value. For example, the body for the concatenate function could be:



Now you can try out your new function. Type in sample parameters into the test data column, for example:



When you press the **Enter** or **Return** key the results column will show the result of your function (or you'll see an error message if you made a mistake). In this case the result is helloworld.



function result displayed here

That's all there is too it! Your custom function is now ready to use in any formula.

Function Names

Function names may consist only of alphabetic characters - no numbers, punctuation or other characters. Upper or lower case is ok, but it is ignored, so you can't create separate functions called **alpha()** and **ALPHA()**. If you rename a custom function the old function will continue to work until you restart Panorama. For compatibility with future versions of Panorama we recommend that you avoid generic names that might be used by a built in Panorama function in the future. If a future version of Panorama uses the name you have chosen, your function will no longer work.

Parameter Names

The name of each parameter must be unique within the body of the function definition. This means you should not use generic names like text or sum. You especially should avoid single letter names like x, y, z, a, b, etc. If you want to make absolutely sure to avoid problems you can put chevrons around the parameters, for example `<number>` or `<text>`. You'll need to use these chevrons in both the **name** and **body** fields.

Advanced Topic: The FDF File

When Panorama starts, it looks inside the Panorama folder for a Functions folder inside the Extensions folder. This folder may contain one or more files with names ending in .fdf. These .fdf files contain the actual custom function definitions. Panorama will automatically open and process these files each time it launches. You can't look at these files directly, but they are generated automatically by the Custom Function wizards.

Advanced Topic : Creating Custom Functions In A Procedure

If you are distributing an application to other users you may want to define functions in the .Initialize procedure of your database, instead of using the wizard. This is possible with the RegisterFunction procedure statement. This statement has four parameters: **folder**, **name**, **parametercount** and **body**.

folder - This parameter is currently unused. For now, simply supply an empty string ("").

name - This is the name of the function. It must be all uppercase letters, terminated by a (. Here are some valid examples:

```
CONCATENATE(
MYCOOLFUNCTION(
```

parametercount - This is the number of function parameters.

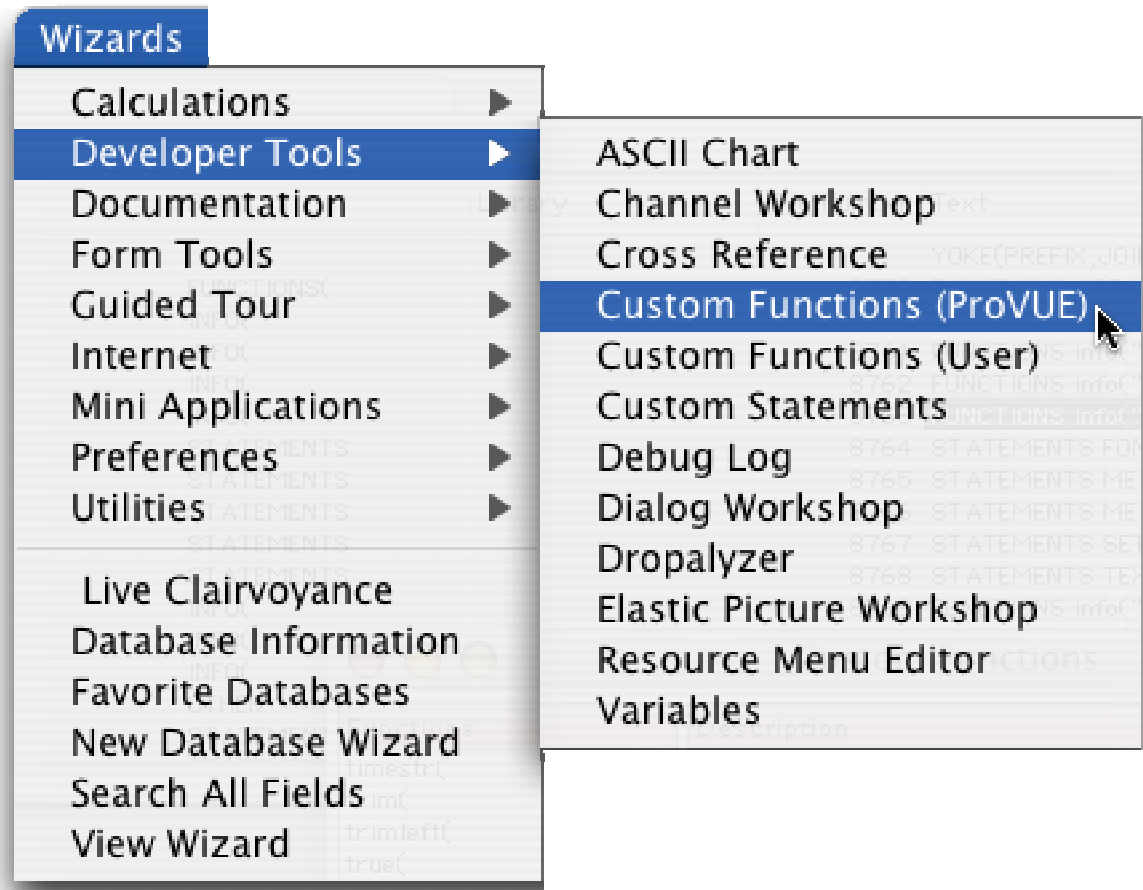
body - This is the the body of the function. Parameters should be represented as **•1**, **•2**, **•3**, etc.

This example shows how to add a custom `concatenate(` function:

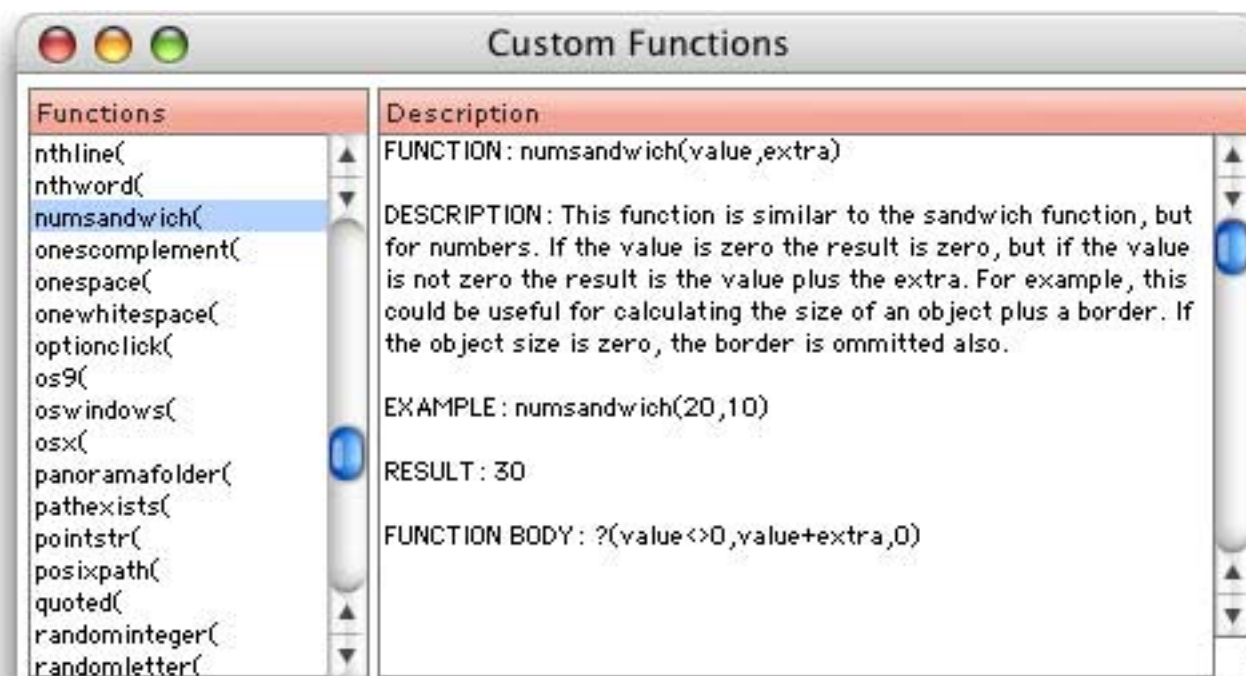
```
registerfunction "", "CONCATENATE(", 2, {•1+•2}
```

The Custom Functions (ProVUE) Wizard

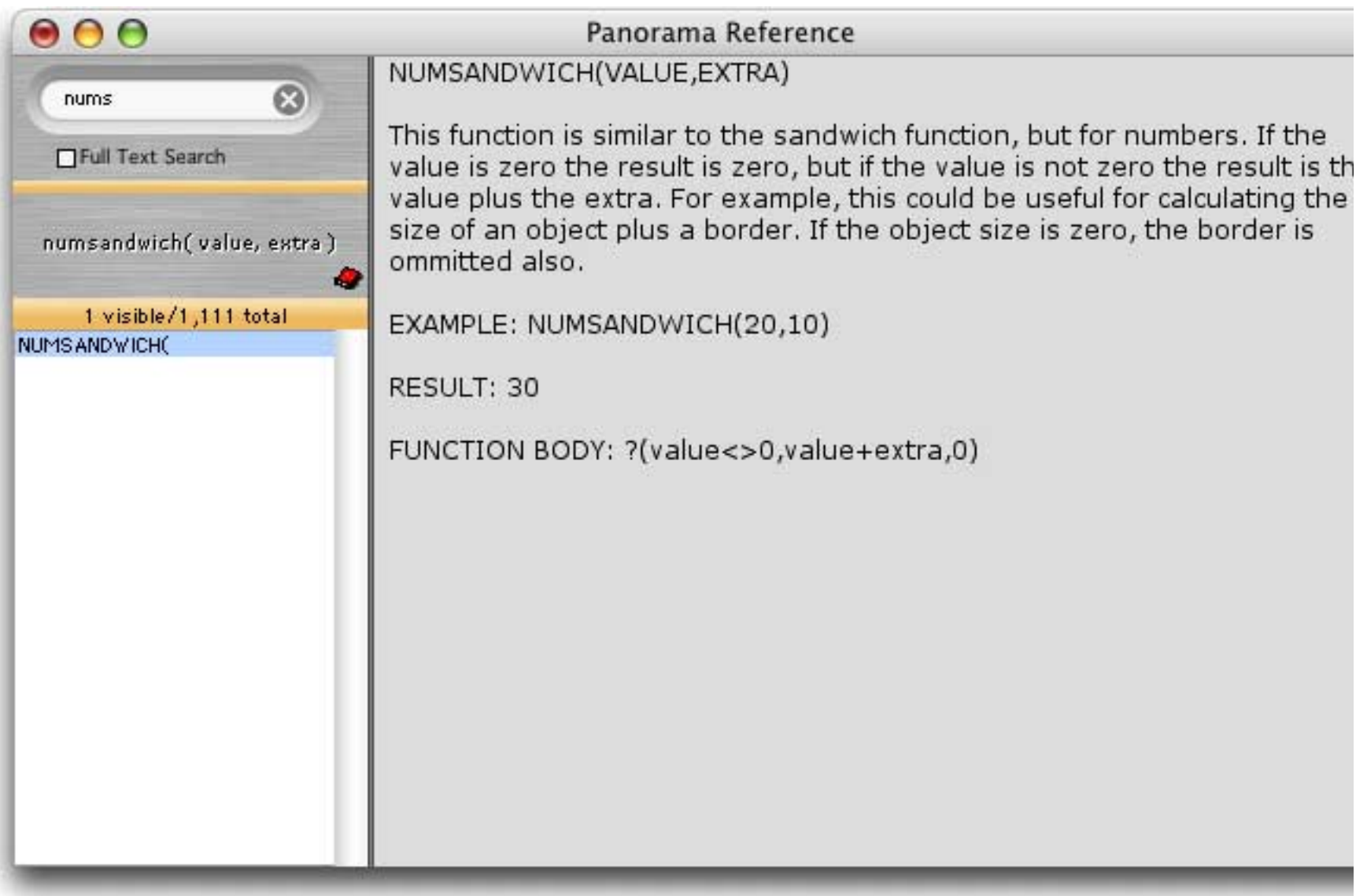
Panorama includes a number of custom functions that have already been defined for you. To see a list of these functions open the **Custom Functions (ProVUE)** wizard.



All of the functions displayed in this wizard are ready to use (and most are already described earlier in this chapter).



You can also find these functions in the **Programming Reference** wizard. It's easy to identify custom functions in this wizard because the information is all displayed in a single, plain font.



If you have any suggestions for additional functions you would like to see appear in this library, please let us know.

Chapter 2: Procedures



Right out of the box, Panorama is a very flexible program. Its built in menus and tools bring incredible power to your fingertips. In spite of this power, however, Panorama is a general purpose tool. To get the most out of Panorama you'll want to customize it to meet the specific needs of your business or industry. Doing this takes an up front investment of time and/or money. But if done properly, the payoff can be huge—a tool optimized specifically for running and organizing your business or life, not someone else's idea of how things should be done. You'll save time, reduce errors, and look more professional to your customers, vendors, employees and/or supervisors.

Programming Isn't Magic!

If you've never programmed before, the idea of programming may seem like magic. But really there's nothing magic about it at all. Programming doesn't really add any new features or capabilities to Panorama. Anything that can be done with programming can also be done manually with Panorama's standard menus and tools.

If programming doesn't add any new features, what is it for? Many database tasks take more than one step to complete. To set up a report, for example, you may need to select certain information, sort the database, and perform calculations. A program allows you to define such a sequence of steps in advance. Once the sequence of steps is defined, you don't have to perform that sequence of steps manually any more. Simply ask the computer and it will perform the steps for you, flawlessly and at the highest possible speed. This lets the computer do what it does best, remember things accurately, and frees your mind for more important tasks. Eventually you can teach Panorama all the everyday tasks you need for running your organization. Of course every journey begins with a single step, so let's get started!

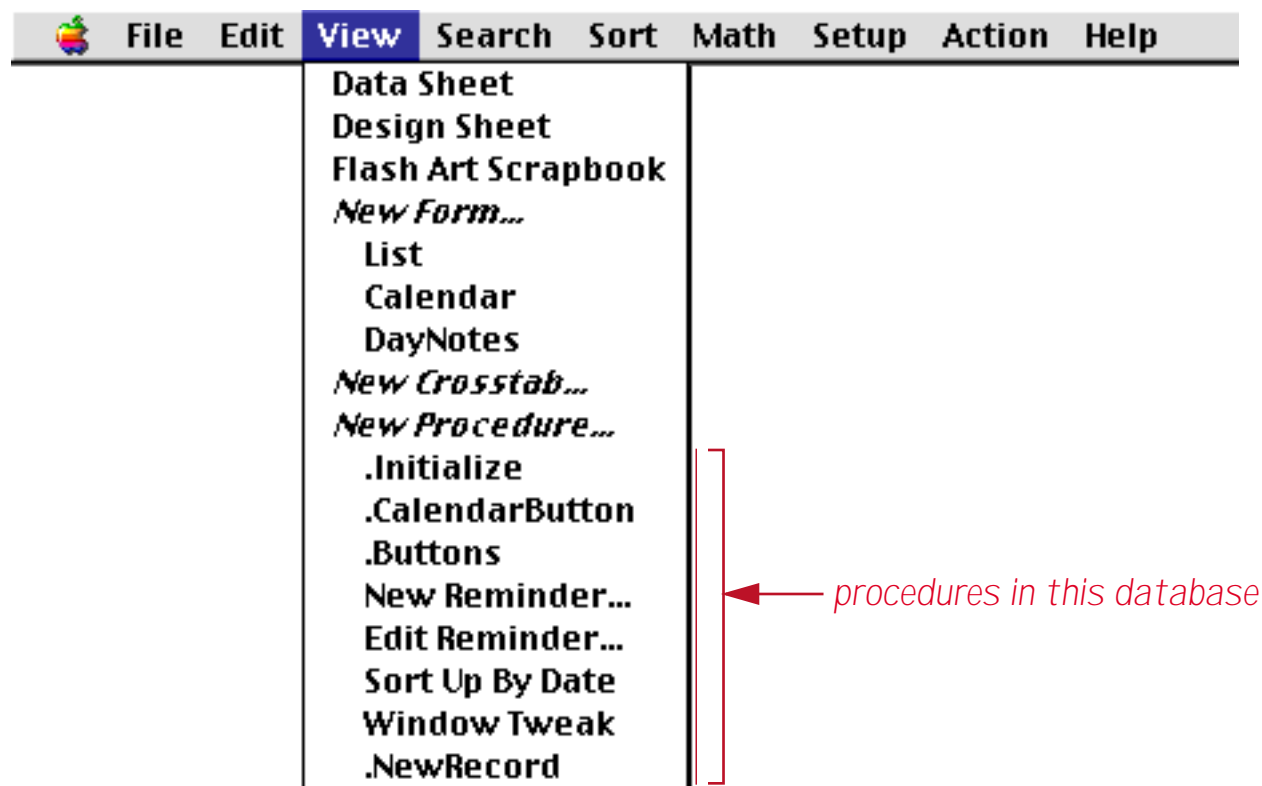
Introduction to (Panorama) Programming

The next few pages introduce the fundamentals of programming with Panorama. You'll learn how complex programs are assembled from a series of small steps, and you'll learn the nuts and bolts of actually creating and using programs. If you are experienced with other programming languages like C or Basic, much of this material will be familiar to you already. If not, welcome to the exciting world of computer programming!

Procedures

A complete program is assembled by combining a series of steps together so that they perform a complete task. In Panorama this complete series of steps is called a procedure. Each procedure performs a complete task from start to finish.

Each database can contain many procedures—one for each task that needs to get done in that database. To help keep all these procedures straight, Panorama requires you to give each procedure a unique name. The procedure name is used whenever you need to refer to the procedure—in a menu, a button, etc. All of the procedures in a database are listed in the **View** menu.



You can view a procedure by selecting it from this menu (“[Switching Between Views](#)” on page 168 of the *Panorama Handbook*). You can also open the procedure in a new window (see “[Opening More Than One Window Per Database](#)” on page 169 of the *Panorama Handbook*) by holding down either the **Control** key (Macintosh) or the **Alt** key (Windows) while you select the procedure from the **View** menu.

Statements

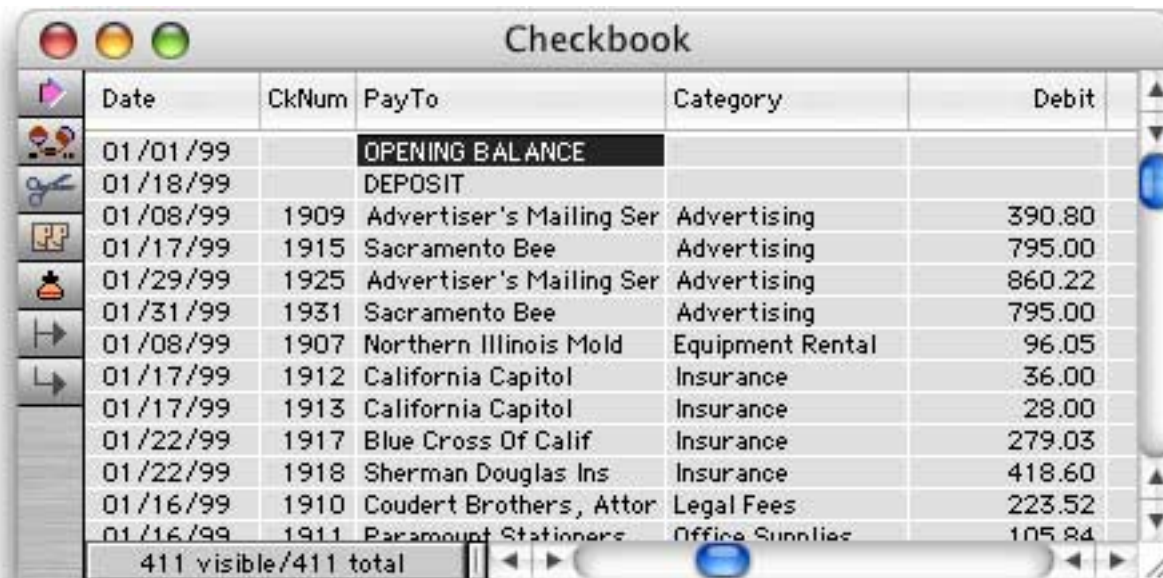
As mentioned in the previous section, a procedure is simply a series of steps. Programmers have a special name for these steps—they call them statements. Each statement is simply a single step to be performed by the computer. Most statements start with a special word (sometimes called a command or keyword) that tells Panorama what the statement should do, for example **SortUp**, **Select**, **Print**, **Open**. Panorama understands several hundred different keywords that perform a wide variety of operations.

A statement may consist of a keyword all by itself, for example **SortUp** or **CloseWindow**. However, many keywords also require additional options, for example **Open "MyDatabase"** or **Select Price>200**. These additional options are called parameters. If a keyword uses parameters, they must follow the keyword in the program.

(Note to experienced programmers: Unlike many programming languages, Panorama’s keywords or commands are not reserved words. This means, for example, that you can use a database column named **Print** or create a variable named **Open**. These names are perfectly OK in Panorama’s programming language. However, using keywords in this way could easily result in programs that are very confusing to read, so we recommend that you avoid keywords when you are defining fields and variables.)

A Simple Procedure in Action

Let's take a look at a simple procedure and see how it works. This procedure is part of a **Checkbook** database that looks like this:



Date	CkNum	PayTo	Category	Debit
01/01/99		OPENING BALANCE		
01/18/99		DEPOSIT		
01/08/99	1909	Advertiser's Mailing Ser	Advertising	390.80
01/17/99	1915	Sacramento Bee	Advertising	795.00
01/29/99	1925	Advertiser's Mailing Ser	Advertising	860.22
01/31/99	1931	Sacramento Bee	Advertising	795.00
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
01/17/99	1912	California Capitol	Insurance	36.00
01/17/99	1913	California Capitol	Insurance	28.00
01/22/99	1917	Blue Cross Of Calif	Insurance	279.03
01/22/99	1918	Sherman Douglas Ins	Insurance	418.60
01/16/99	1910	Coudert Brothers, Attor	Legal Fees	223.52
01/16/99	1911	Paramount Stationers	Office Supplies	105.84

The procedure itself contains six statements. In this procedure each statement is on its own line, but as you'll see later this is not necessary.



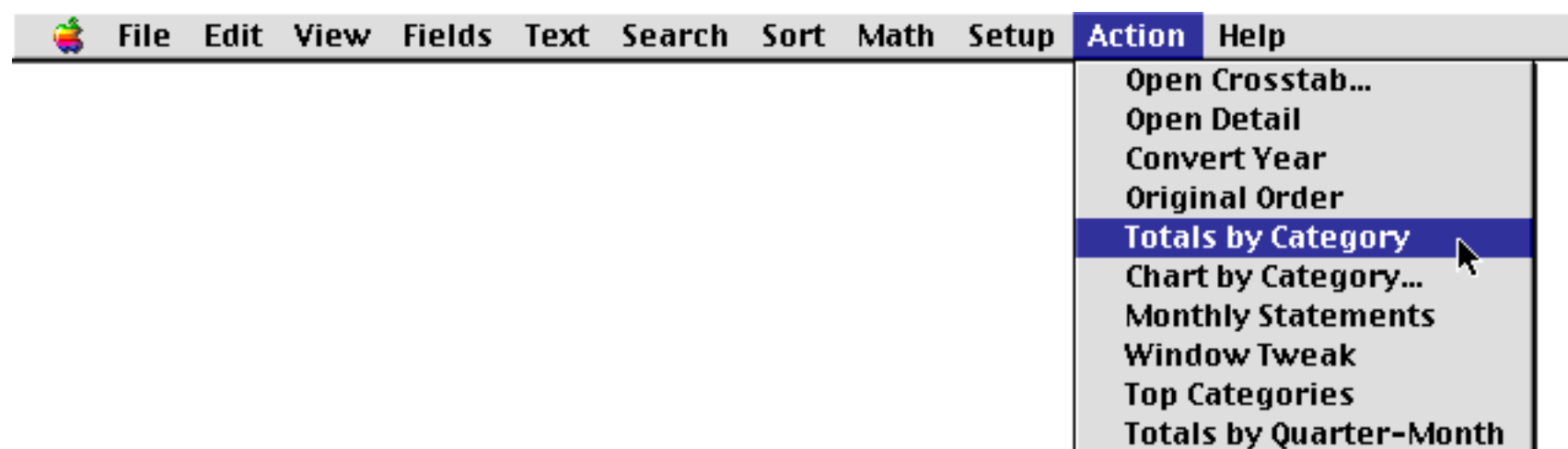
```

select Debit>0
field Category
groupup
field Debit
total
outlinelevel 1

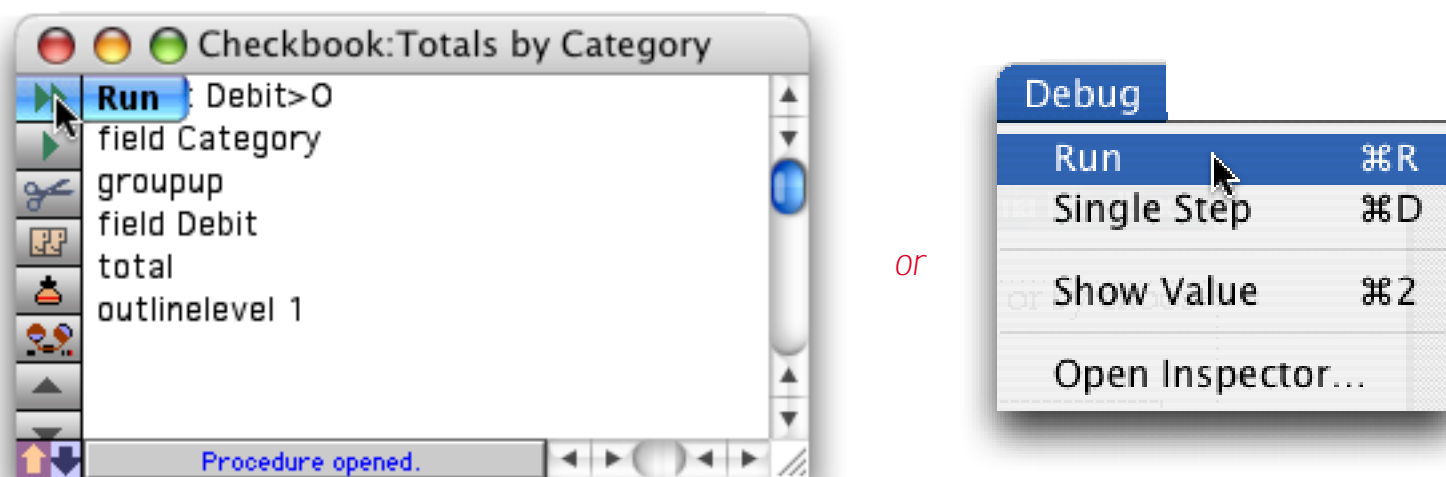
```

Procedure opened.

To use a procedure you need to start it somehow. (This is called “triggering” the procedure.) There are two easy ways to do this. If the data sheet is currently the top window you can trigger the procedure by choosing it from the **Action** menu.



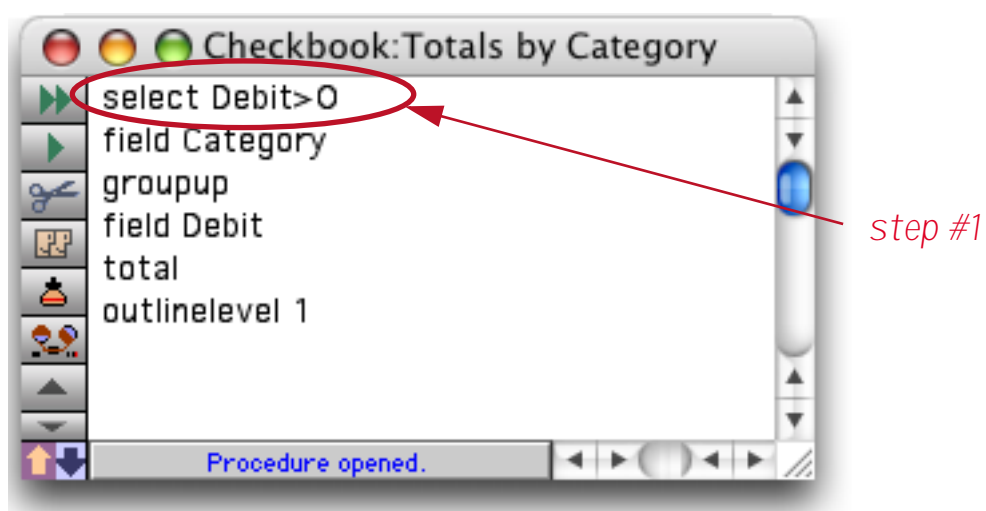
If the procedure itself is currently the top window you can trigger it by clicking on the **Run** tool, or by choosing **Run** from the **Debug** menu.



Once the procedure is triggered Panorama begins performing the steps in the procedure. You can watch as Panorama rapidly performs each step, kind of like a fast action “Keystone Kops” movie.

Let’s take a look at how Panorama performs each of the steps in our sample procedure, starting with step number 1.

```
select Debit>0
```



This first statement selects a subset of the data. Panorama performs this step exactly as if you had chosen the **Find/Select** command (see “[The Find/Select Dialog](#)” on page 336 of the *Panorama Handbook*) and filled in the dialog like this. (However since Panorama already knows what to do it doesn’t actually display this dialog.)



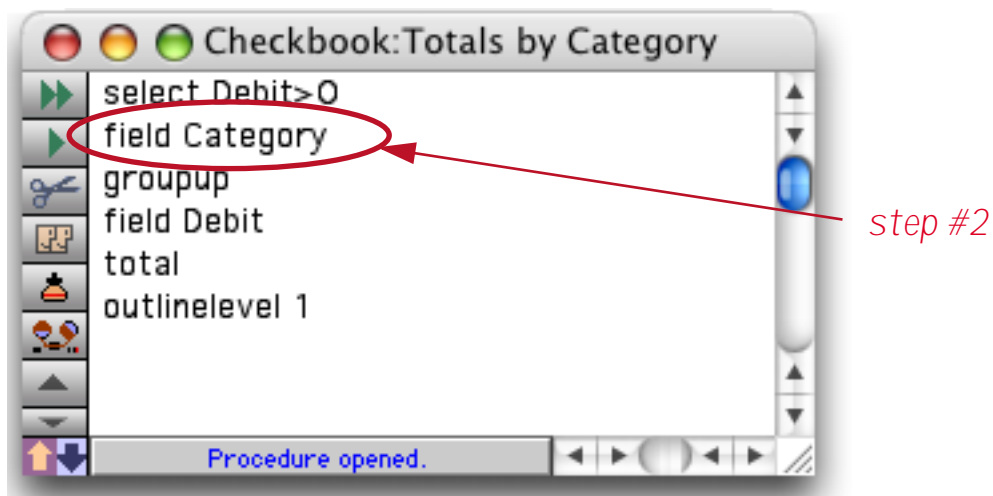
The result of this step is that a subset of the database is now selected.

Date	CkNum	PayTo	Category	Debit
01/08/99	1909	Advertiser's Mailing Ser	Advertising	390.80
01/17/99	1915	Sacramento Bee	Advertising	795.00
01/29/99	1925	Advertiser's Mailing Ser	Advertising	860.22
01/31/99	1931	Sacramento Bee	Advertising	795.00
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
01/17/99	1912	California Capitol	Insurance	36.00
01/17/99	1913	California Capitol	Insurance	28.00
01/22/99	1917	Blue Cross Of Calif	Insurance	279.03
01/22/99	1918	Sherman Douglas Ins	Insurance	418.60
01/16/99	1910	Coudert Brothers, Attor	Legal Fees	223.52
01/16/99	1911	Paramount Stationers	Office Supplies	105.84
01/22/99	1919	Cannon Astro	Office Supplies	145.72
01/25/99	1921	Nebs	Office Supplies	77.27

393 visible/411 total

On to step number 2.

field Category



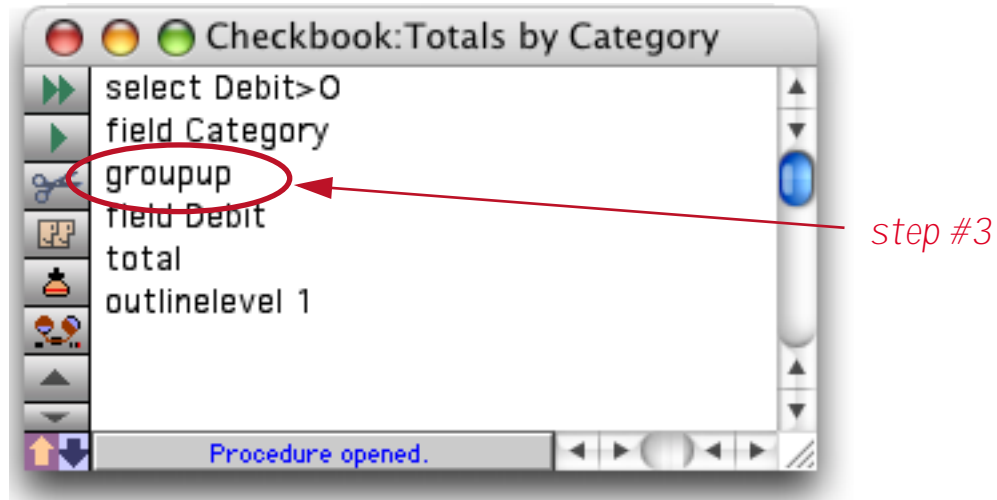
Many Panorama operations require you to click on a field before you perform an operation. For example, to sort or group by a particular column you would first click anywhere in the column. In a procedure this is accomplished with the **field** statement. If you watch quickly you'll see the cursor jump over to the **Category** column as Panorama performs this step.

Date	CkNum	PayTo	Category	Debit
01/08/99	1909	Advertiser's Mailing Ser	Advertising	390.80
01/17/99	1915	Sacramento Bee	Advertising	795.00
01/29/99	1925	Advertiser's Mailing Ser	Advertising	860.22
01/31/99	1931	Sacramento Bee	Advertising	795.00
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
01/17/99	1912	California Capitol	Insurance	36.00
01/17/99	1913	California Capitol	Insurance	28.00
01/22/99	1917	Blue Cross Of Calif	Insurance	279.03
01/22/99	1918	Sherman Douglas Ins	Insurance	418.60
01/16/99	1910	Coudert Brothers, Attor	Legal Fees	223.52
01/16/99	1911	Paramount Stationers	Office Supplies	105.84
01/22/99	1919	Cannon Astro	Office Supplies	145.72
01/25/99	1921	Nebs	Office Supplies	77.27

393 visible/411 total

Now Panorama is ready for step number 3.

groupup



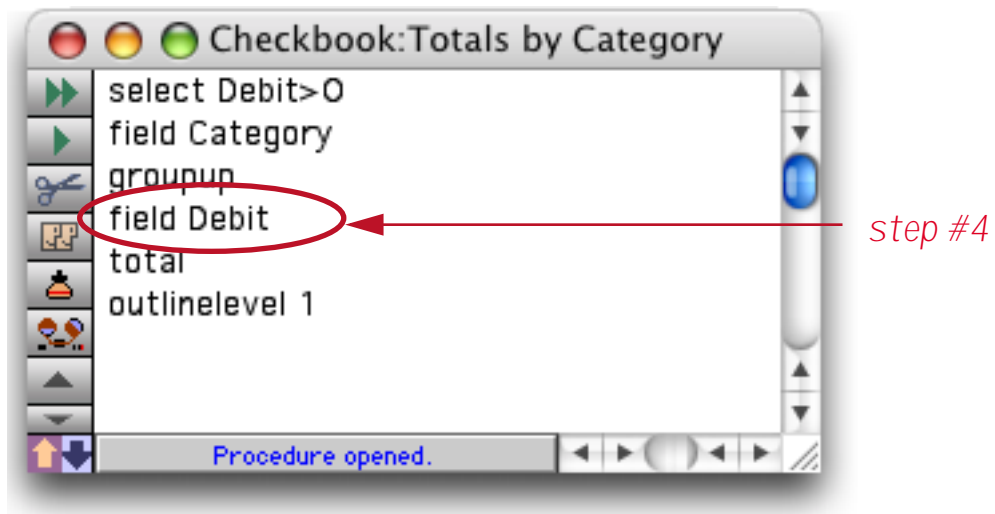
This statement tells Panorama to group the database, just as if you had selected **Group Up** from the Math menu (see "[STEP 1 - GROUP](#)" on page 394 of the *Panorama Handbook*).

The screenshot shows a window titled "Checkbook" displaying a table of transactions. The table is grouped by category, with "Advertising" and "Auto" categories highlighted in bold. The status bar at the bottom indicates "410 visible/429 total".

Date	CkNum	PayTo	Category	Debit
09/28/99	2299	Advertiser's Mailing Ser	Advertising	167.00
09/28/99	2298	Graphic Depot	Advertising	344.00
Advertising				
02/01/99	1938	Unocal	Auto	182.59
02/09/99	1968	Unocal	Auto	57.62
03/16/99	2007	Unocal	Auto	33.32
05/24/99	2111	Unocal	Auto	119.05
07/16/99	2189	Unocal	Auto	38.11
07/24/99	2213	Unocal	Auto	34.44
08/20/99	2240	Unocal	Auto	89.91
Auto				
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
02/09/99	1950	Pitney Bowes	Equipment Rental	73.14

On to step number 4.

field Debit



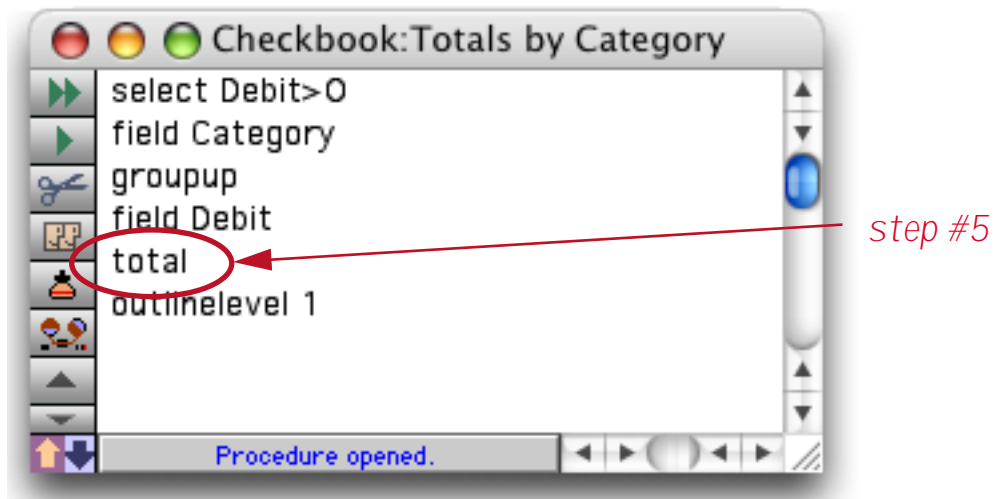
If we were performing this sequence of steps manually we would now click on the **Debit** column.

Date	CkNum	PayTo	Category	Debit
09/28/99	2299	Advertiser's Mailing Ser	Advertising	167.00
09/28/99	2298	Graphic Depot	Advertising	344.00
Advertising				
02/01/99	1938	Unocal	Auto	182.59
02/09/99	1968	Unocal	Auto	57.62
03/16/99	2007	Unocal	Auto	33.32
05/24/99	2111	Unocal	Auto	119.05
07/16/99	2189	Unocal	Auto	38.11
07/24/99	2213	Unocal	Auto	34.44
08/20/99	2240	Unocal	Auto	89.91
Auto				
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
02/09/99	1950	Pitney Bowes	Equipment Rental	73.14

410 visible / 429 total

Next is step number 5.

total



This statement is the same as choosing the Total command from the Math menu (see "[Total](#)" on page 398 of the *Panorama Handbook*).

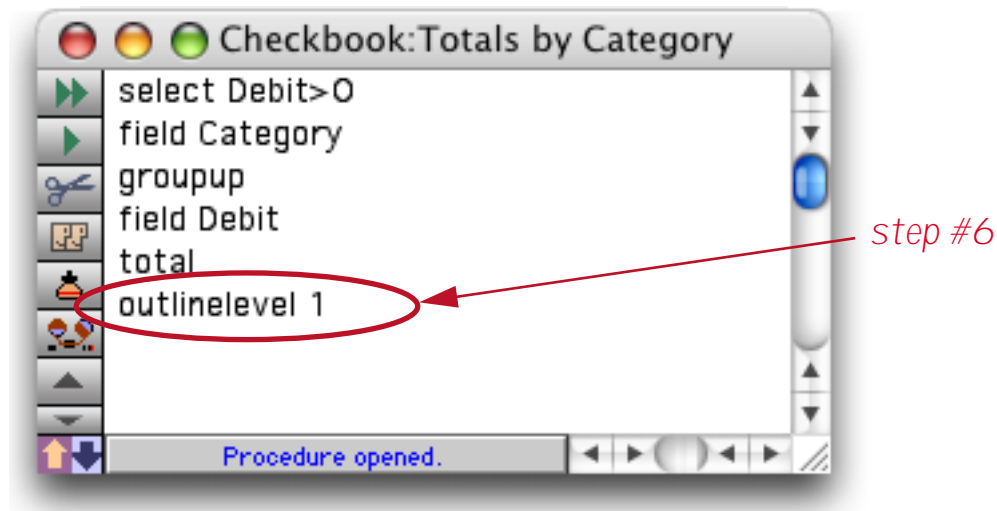
The screenshot shows the "Checkbook" window with a table of transactions. The table has columns for Date, CkNum, PayTo, Category, and Debit. The 'Advertising' and 'Auto' category totals are circled in red.

Date	CkNum	PayTo	Category	Debit
09/28/99	2299	Advertiser's Mailing Ser	Advertising	167.00
09/28/99	2298	Graphic Depot	Advertising	344.00
			Advertising	34,516.82
02/01/99	1938	Unocal	Auto	182.59
02/09/99	1968	Unocal	Auto	57.62
03/16/99	2007	Unocal	Auto	33.32
05/24/99	2111	Unocal	Auto	119.05
07/16/99	2189	Unocal	Auto	38.11
07/24/99	2213	Unocal	Auto	34.44
08/20/99	2240	Unocal	Auto	89.91
			Auto	555.04
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
02/09/99	1950	Pitney Bowes	Equipment Rental	73.14

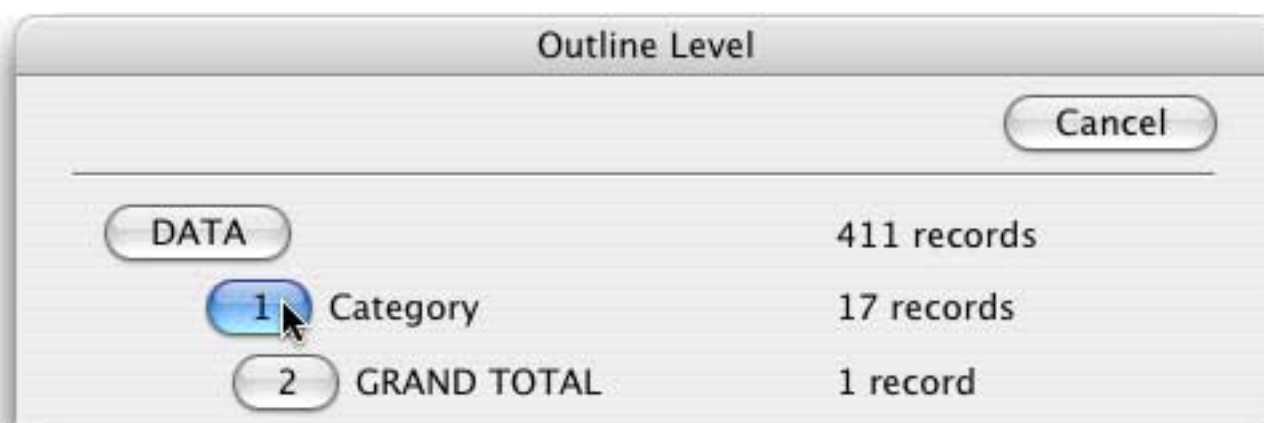
410 visible/429 total

And now for the final step, number 6.

```
outlinelevel 1
```



This statement is the same as choosing the **Outline Level** command from the Sort menu (see “[Sorting by Summary Value](#)” on page 406 of the *Panorama Handbook*). The parameter on this statement, **1**, tells Panorama to simulate pressing the **1** button in the dialog. (Once again, since Panorama already knows what to do it doesn’t actually display the dialog.)



Here’s the final result after all six statements have completed.

Date	CkNum	PayTo	Category	Debit
			Advertising	34,516.82
			Auto	555.04
			Equipment Rent:	632.01
			Fixed Assets	9,774.47
			Insurance	8,234.53
			Legal Fees	1,288.07
			Maintenance	3,673.99
			Office Supplies	7,261.09
			Postage	1,456.35
			Printing	188.96
			Purchases	66,217.17
			Rent	35,026.34
			Shipping	1,928.20

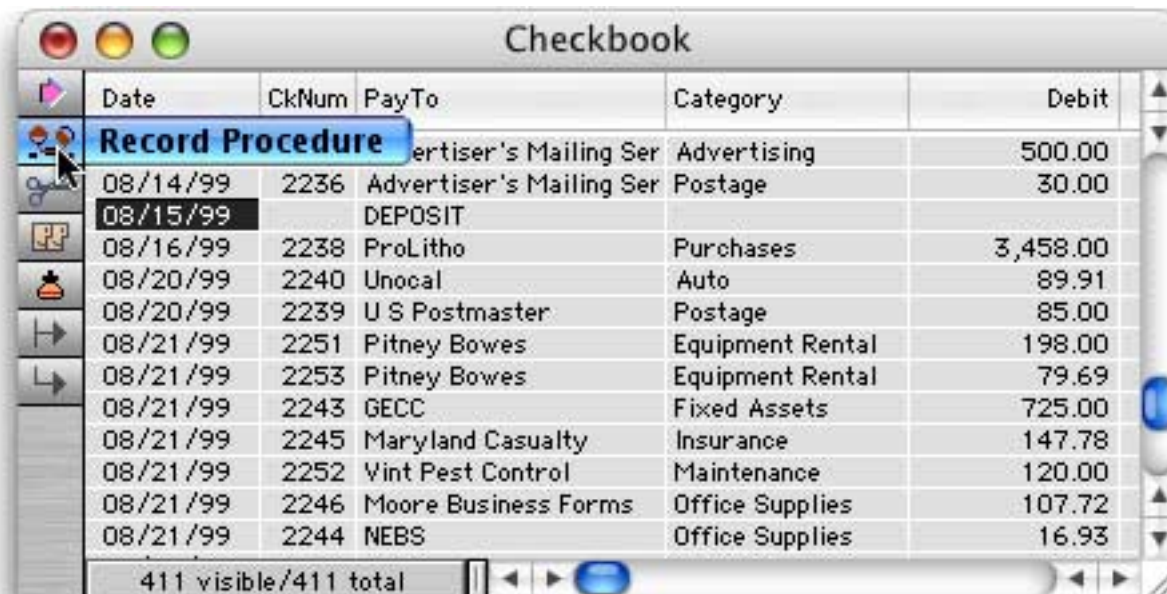
17 visible / 429 total

Pretty simple, is it not? This is the basic operation of any procedure — first the procedure is triggered, then Panorama performs each statement from top to bottom. (Later you’ll learn several ways to change the top to bottom order when it is necessary, see “[Control Flow](#)” on page 255.)

Creating a Procedure with the Recorder

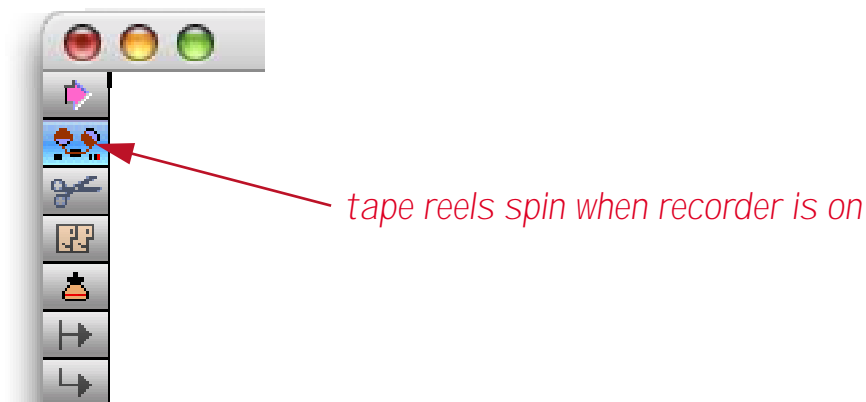
A basic procedure like the one in the last section is very easy to create with Panorama's built in procedure recorder. The procedure recorder is like a tape recorder that records your actions as you work. Recording a procedure is a four step process— 1) start the recorder, 2) perform the steps while Panorama records, 3) stop the recorder, and 4) give the new procedure a name.

To start the procedure recorder, click the **Record Procedure** tool (available in the Data Sheet and Form tool palettes (unless you are in Graphics Mode)).



Date	CkNum	PayTo	Category	Debit
08/14/99	2236	Advertiser's Mailing Ser	Advertising	500.00
08/14/99	2236	Advertiser's Mailing Ser	Postage	30.00
08/15/99		DEPOSIT		
08/16/99	2238	ProLitho	Purchases	3,458.00
08/20/99	2240	Unocal	Auto	89.91
08/20/99	2239	U S Postmaster	Postage	85.00
08/21/99	2251	Pitney Bowes	Equipment Rental	198.00
08/21/99	2253	Pitney Bowes	Equipment Rental	79.69
08/21/99	2243	GECC	Fixed Assets	725.00
08/21/99	2245	Maryland Casualty	Insurance	147.78
08/21/99	2252	Vint Pest Control	Maintenance	120.00
08/21/99	2246	Moore Business Forms	Office Supplies	107.72
08/21/99	2244	NEBS	Office Supplies	16.93

The “reels” on the recorder tool will begin to spin. This lets you know that Panorama is recording your actions.



Once the recorder is running just continue to use Panorama normally. Panorama will record every menu command or tool that you use. To demonstrate the recorder we'll create a short procedure that calculates the grand total. Start by clicking anywhere in the **Debit** field.

Date	CkNum	PayTo	Category	Debit
08/14/99	2237	Advertiser's Mailing Ser	Advertising	500.00
08/14/99	2236	Advertiser's Mailing Ser	Postage	30.00
08/15/99		DEPOSIT		
08/16/99	2238	ProLitho	Purchases	3,458.00
08/20/99	2240	Unocal	Auto	89.91
08/20/99	2239	U S Postmaster	Postage	85.00
08/21/99	2251	Pitney Bowes	Equipment Rental	198.00
08/21/99	2253	Pitney Bowes	Equipment Rental	79.69
08/21/99	2243	GECC	Fixed Assets	725.00
08/21/99	2245	Maryland Casualty	Insurance	147.78
08/21/99	2252	Vint Pest Control	Maintenance	120.00
08/21/99	2246	Moore Business Forms	Office Supplies	107.72
08/21/99	2244	NEBS	Office Supplies	16.93

Next choose **Total** from the Math menu. Panorama calculates the total.

Date	CkNum	PayTo	Category	Debit
09/19/99	2288	MCI	Telephone	67.59
09/19/99	2290	City Of Caboose	Utilities	103.15
09/19/99	2291	S C E	Utilities	81.13
09/19/99	2292	So. Calif. Gas Co.	Utilities	154.95
09/21/99	2294	Advertiser's Mailing Ser	Postage	167.00
09/21/99	2295	Advertiser's Mailing Ser	Postage	67.00
09/26/99	2297	AC Label Company	Advertising	205.97
09/26/99	2296	TesLabe	Fixed Assets	2,465.00
09/28/99	2299	Advertiser's Mailing Ser	Advertising	167.00
09/28/99	2298	Graphic Depot	Advertising	344.00
08/23/04	2310			183,651.22

Our simple procedure is complete. To stop the recorder, click on the **Record Procedure** tool again. The “reels” will rewind and stop and a dialog box appears. This dialog box allows you to give your new procedure a name, up to 25 characters. Pick a name that will help you remember what the procedure does, for example **Print Invoices** or **Balance Checkbook**.

Save Recording

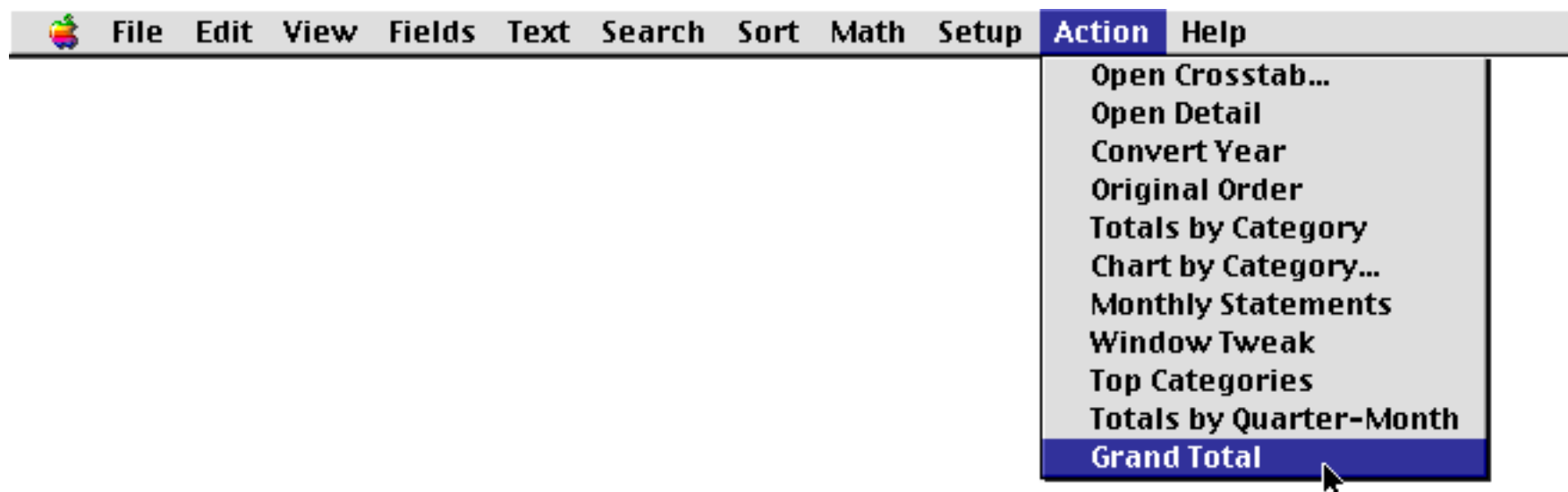
New procedure name...

Grand Total

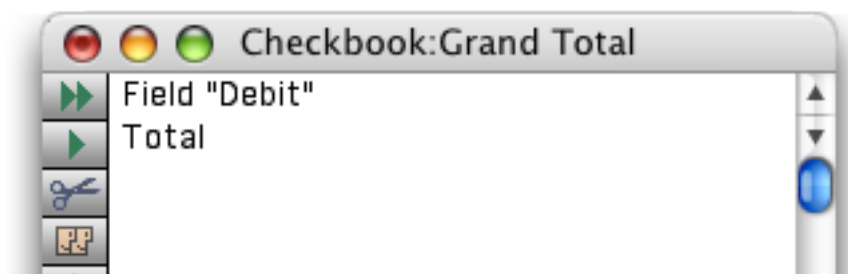
Once you have entered the name, press **Save Procedure**.

Tip: There are five characters you should not use in a procedure name unless you know what you are doing. The five characters are: ^ ; < (/. Later in this manual you'll learn how these characters can be used to create special effects in the **Action Menu** (see "[Action Menu Options](#)" on page 356).

Once you have given the recording a name, the new procedure is added to the end of the **Action Menu**. (If the database doesn't already have a **Action Menu**, it will be created.)



You can play back your new procedure at any time by selecting it from the **Action Menu**. To view the statements in the new procedure you can open it with the **View** menu (hold down the **Control** key (Mac) or **Alt** key (PC) to open the procedure in its own new window.)



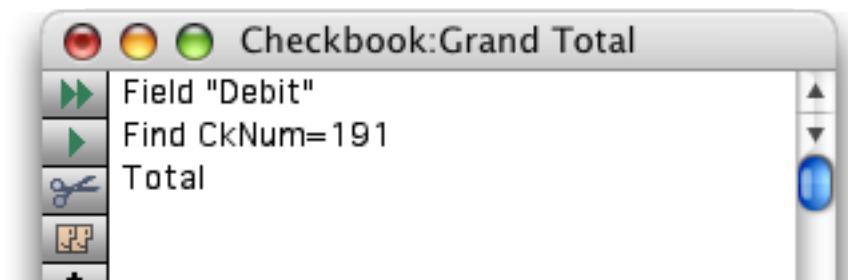
Our simple recording contains two steps. You can use the procedure “as is” or you can customize it further.

Recording Mouse Clicks

Panorama does not normally consider clicking the mouse to be a step—clicking the mouse is not a menu command or tool. However, if you click on a different window or a different field within the current window, Panorama will record that step. The recorder will ignore all other mouse clicks (for example, clicking on the scroll bar, dragging a window to a new position or changing the size of a window, etc.).

Panorama never records the row position of a click. If you look back at the previous section you'll notice that I clicked on check number **1915** when I recorded the procedure. However, this information was not recorded. When the procedure is played back Panorama will not move to check number **1915**, but will stay on whatever check it is already on. The **total** statement doesn't care as long as the column (field) is correct.

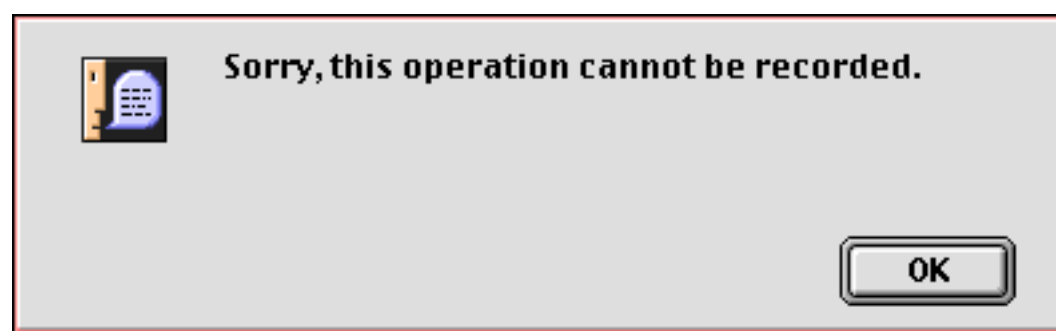
If you do want Panorama to move to a specific record you'll need to use the `find` statement (see "[FIND](#)" on page 5245 of the *Panorama Reference*). You can either create this during recording using the [Find/Select](#) dialog (see "[The Find/Select Dialog](#)" on page 336 of the *Panorama Handbook*) or simply by typing it into the procedure window. Here's a modified version of the procedure that moves to check number 1915.



Of course this example doesn't make much sense because Panorama will only be on check number 1915 for a fraction of a second before the `total` statement makes Panorama jump to the end of the database.

Non Recordable Menus and Tools

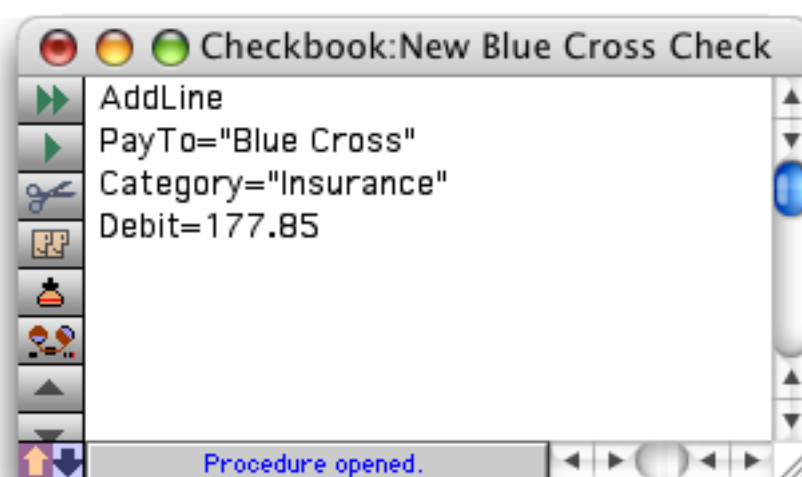
A couple of pages ago we told you that Panorama records every menu and tool when the recorder is on. Sorry, but that was a lie. Some menus and tools are not recordable. Most commands and tools that work with graphics cannot be used. For example, **Sort Up**, **Insert Record**, and **Print** can be used as steps in a procedure, but **Bring to Front**, **Oval**, and **Align** cannot. In general, only actions that affect data can be included in a procedure. If the recorder is on and you attempt to use a menu command or tool that cannot be used as a step in a procedure, Panorama will alert you.



As long as you stick with the Data Sheet and Data Access Mode Forms you'll usually be fine.

Recording Data Entry

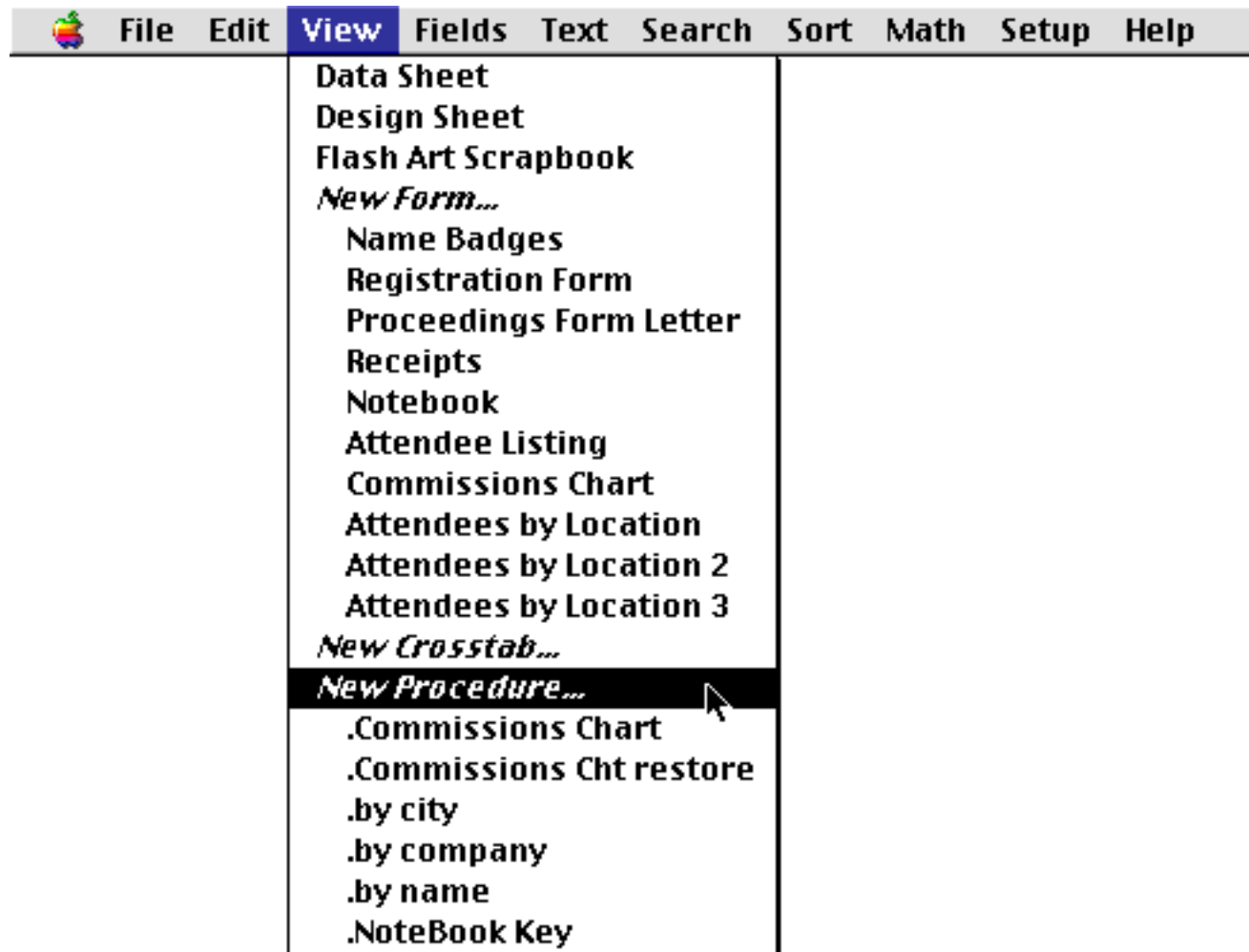
Panorama doesn't do a very good job of recording data entry. Instead of using the recorder we recommend that you use assignment statements to perform data entry, as shown in this example.



You cannot record these assignment statements, you have to type them in manually. See "[Assignment Statements](#)" on page 243 for more information on assignment statements.

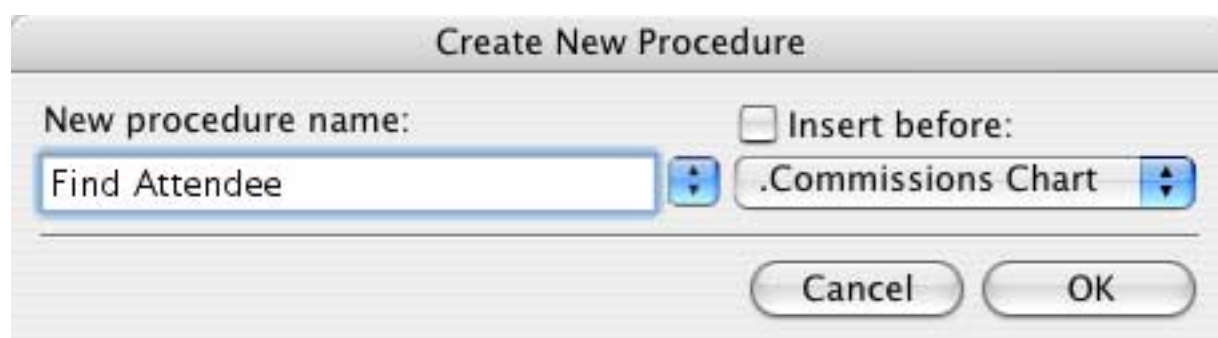
Writing a Procedure from Scratch

If you want to write a procedure from scratch (instead of using the recorder), the first step is to create a new, empty procedure. To do this select **New Procedure** from the **View** menu.



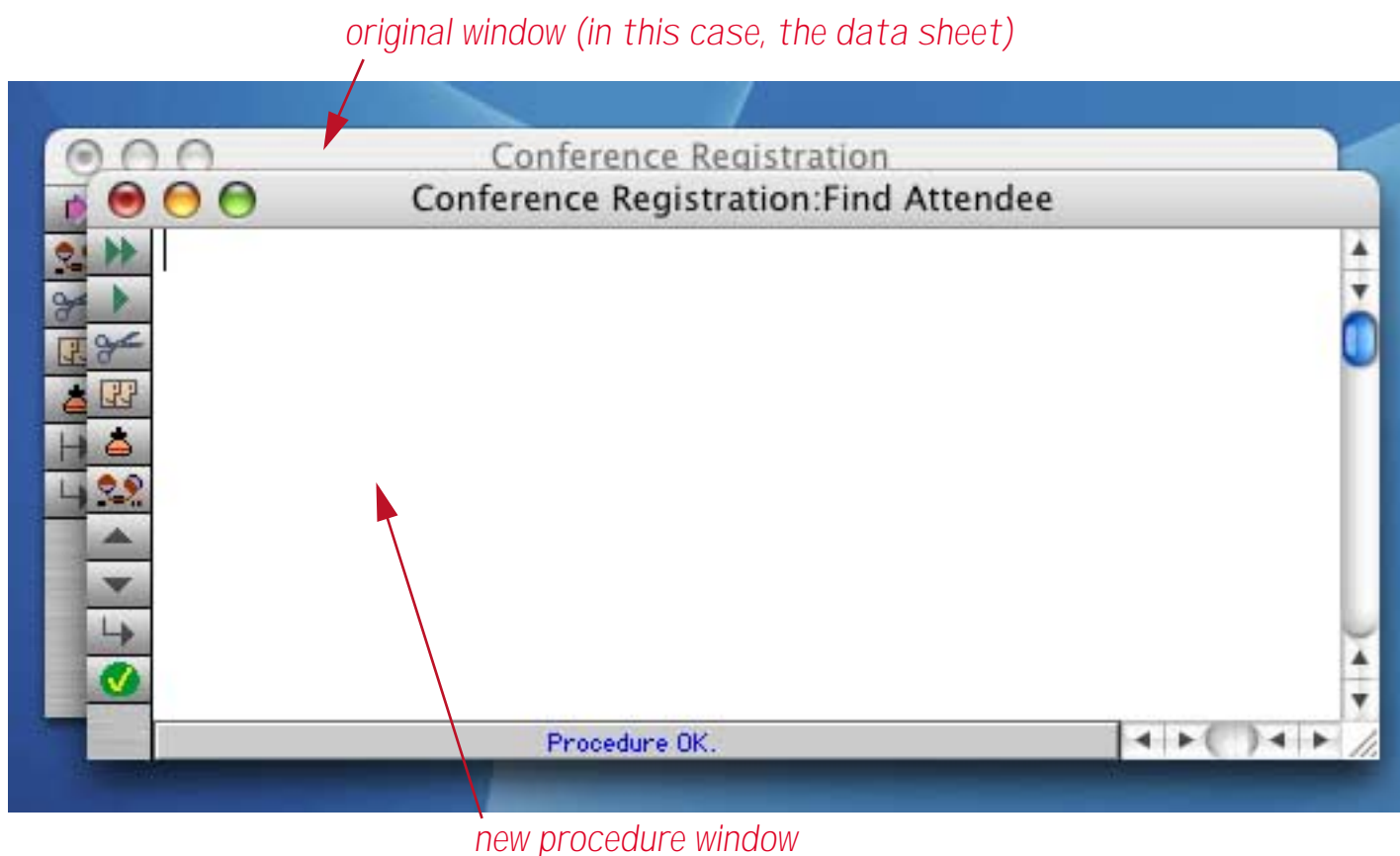
Selecting **New Procedure** normally creates the new procedure in the same window you are currently in. It's often convenient to create the new procedure in a new, separate window, leaving the original window open. That way you can easily flip back and forth between your database window (data sheet or form) and the procedure window. To open the new procedure in a separate window hold down the **Control** key (Macintosh) or **Alt** key (Windows) as you click on the **View** menu.

After you select **New Procedure** this dialog appears.



Type in the name of the new procedure, then press **OK**. The name may be up to 25 characters long, and must be unique within this database. Tip: There are five characters you should not use in a procedure name unless you know what you are doing. The five characters are: ^ ; < (/. Later in this manual you'll learn how these characters can be used to create special effects in the **Action Menu** (see "[Action Menu Options](#)" on page 356).

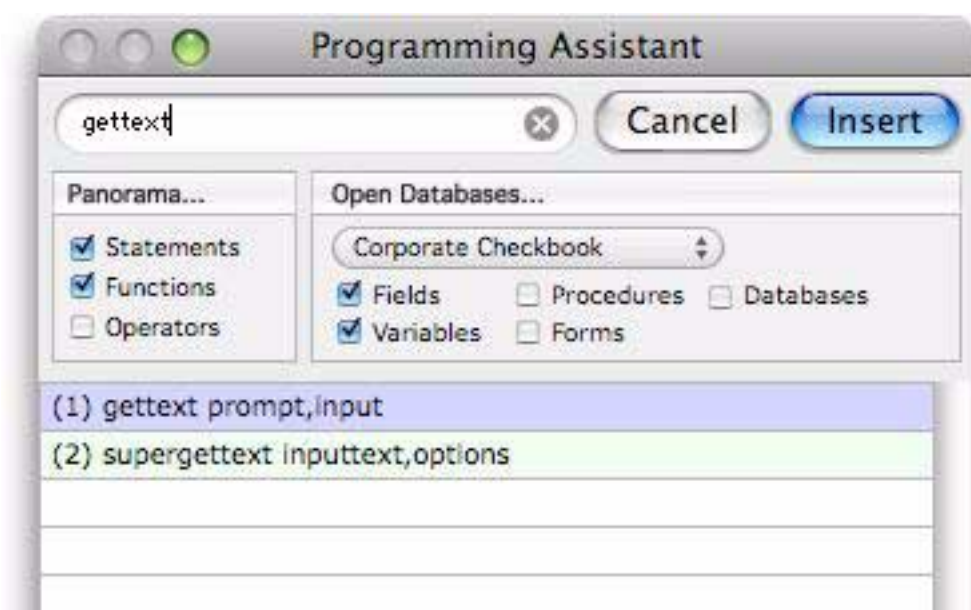
After you press the **OK** button a new, empty procedure is created, and the window switches to show you this new procedure. If you held down the **Control** key (Mac) or **Alt** key (Windows) the new procedure will open in a new window just below and to the right of the original window, like this.



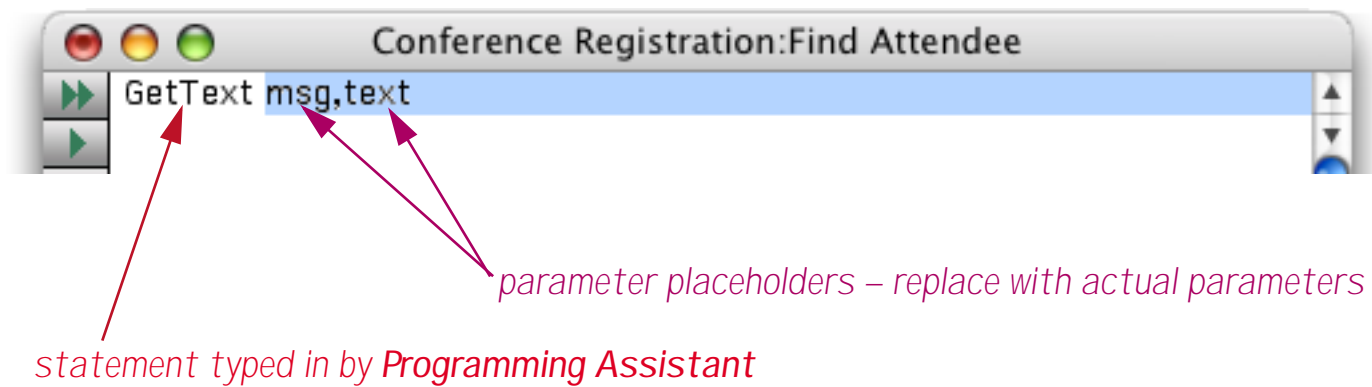
Writing Statements

Once you have created an empty procedure you can start adding statements to the procedure. If you know the keywords for the statements you want to use, just type them in. For example, if you want to sort the data-base in ascending order just type in the statement **SortUp**. By the way, the capitalization of statements doesn't matter, so you could also type **sortup**, **SORTUP**, or even **SoRTuP**. However, keywords are always a single word with no blanks, so **sort up** will not work.

If you don't know the exact keywords for the operations you want the procedure to perform, you have a several choices. You could look up the keyword in this manual, then type it in from the keyboard. You could use the **Programming Assistant** dialog (see "[The Programming Assistant Dialog](#)" on page 225), or you could locate the keyword using the **Topic** submenu of the Programming Context Menu (right click, see "[Topics Submenu](#)" on page 234). Another choice is to use the **Programming Reference** wizard, see "[Programming Reference Wizard](#)" on page 237.



You'll still have to type in any options and parameters yourself, but the **Programming Assistant** will type in placeholders for these values to help remind you that they are necessary (we'll talk more about options and parameters later).



Here is the finished statement with the actual parameters typed in.



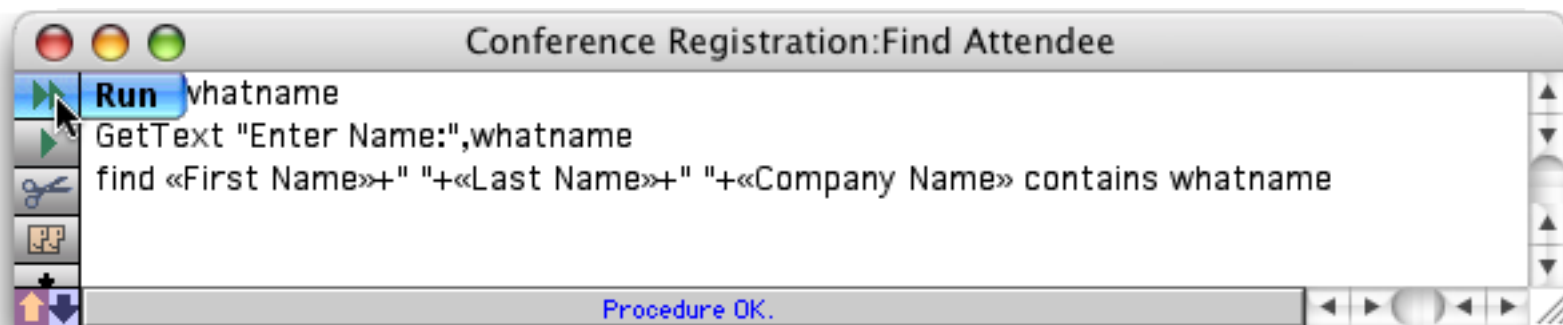
Here is the completed procedure.



This procedure has three statements. The **local** statement creates a local variable named **whatname**. The **gettext** statement displays a dialog asking to enter a name. Whatever is typed in is placed in the **whatname** variable. The **find** statement searches the database to locate the requested name.

Trying Out a Procedure

The easiest way to try out a procedure you are working on is to press the **Run** button (you can also press **Command-R** on a Mac or **Control-R** on a PC).



When you run a procedure this way Panorama will automatically switch to a window that contains data. Any window that allows you to display and edit data will do (as long as it is in the same database), so Panorama will pick the topmost one that will work. If there is no window that will work (for example if the procedure is in the only window for this database), Panorama will display an error message and the procedure will not run.

In this case Panorama will switch to the data sheet window and begin performing the steps in the procedure, starting with the topmost statement.

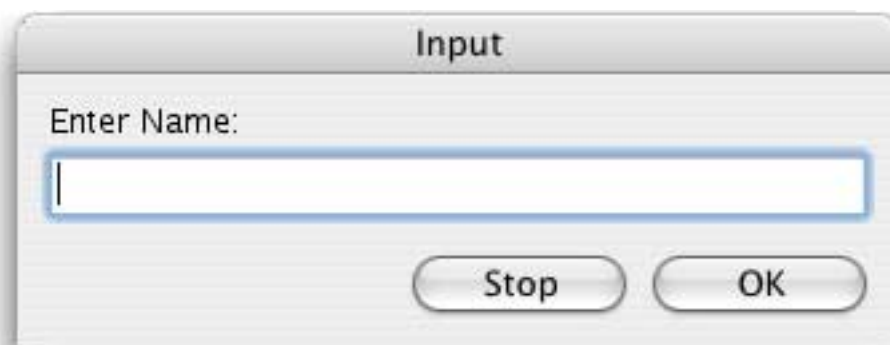
```
local whatname
```

This statement allocates a local variable named `whatname` for temporary storage (see “[Variables](#)” on page 53). This operation is completely invisible.

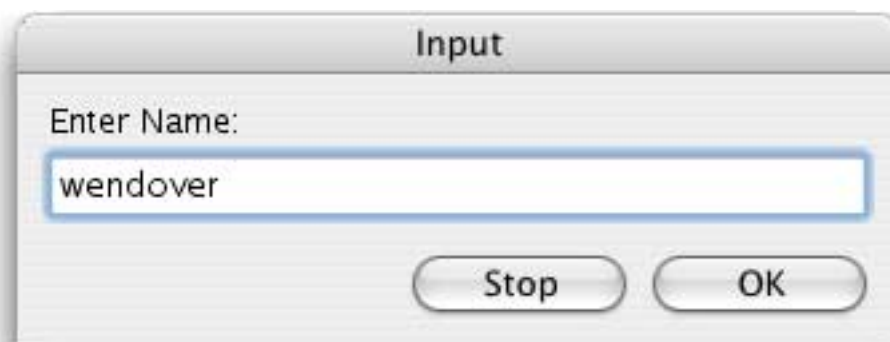
On to step 2 —

```
GetText "Enter Name:",whatname
```

This statement displays a dialog asking the user to type in some text.



Enter the name you want to search for.



When you press **OK** the dialog disappears and the procedure continues on with the next statement. Before it does, however, it copies the text that was typed in into the variable named `whatname`.

```
find <First Name>+" "+<Last Name>+" "+<Company Name> contains whatname
```

This final statement searches three fields in the database to see if they contain whatever text was typed in (see “[Comparison Operators](#)” on page 124). This database does contain the name **wendover**, so Panorama jumps to that record. The name is circled in the illustration below to make it more clear.

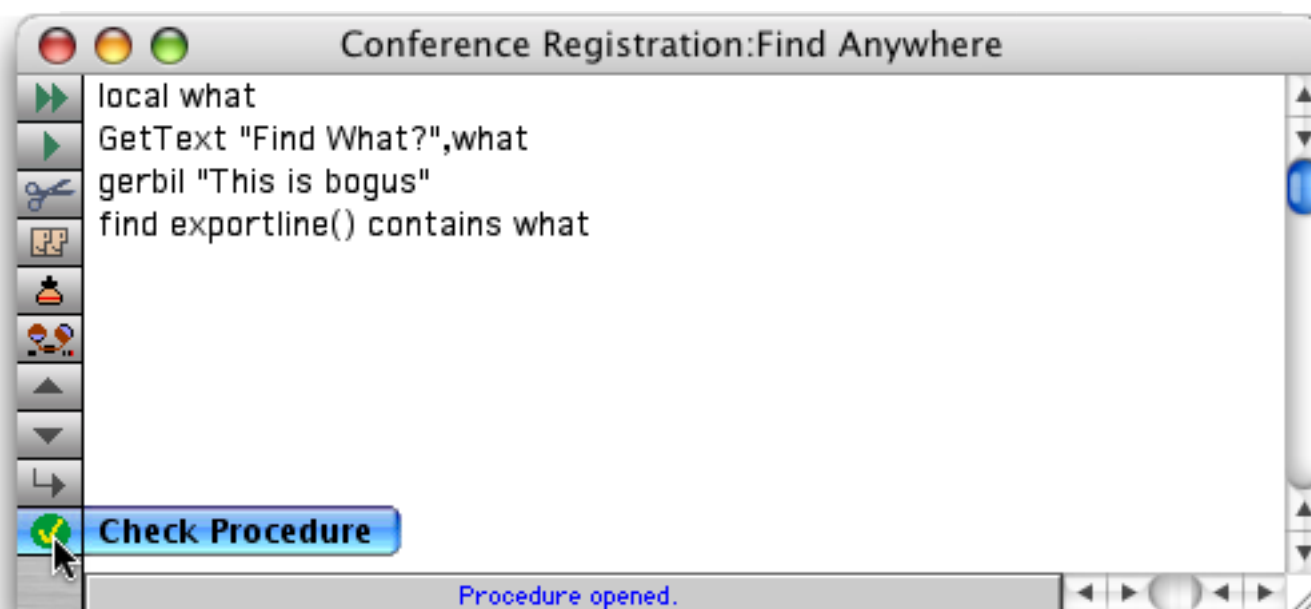
T	First Name	Last Name	Company Name	Street Address	Suite Box	City	Stat
	Ms. Christy	Alpert	Signal Research	1120 Sharon Pa		Cupertino	CA
	Mr. Arthur	Clairmont	South Coast Office Produc	4390 Kaiser Dr		Cupertino	CA
	Mr. Harold	Cobb	Cobb Associates	3912 Phillip St.		Cupertino	CA
	Mrs. Sherry	Grossman	Pablo Distribution	1400 Valley Riv		Cupertino	CA
	Mr. Peter	Parks	Hamilton Press	41 Kenosia Ave		Cupertino	CA
	Mrs. Michelle	Adams	Sceptre	10159 Alliance		Cupertino	CA
	Mrs. Kathy	Schwartz	Wendover Insurance Grou	814 Castro St.		Long Beach	CA
	Mr. Charles	Arrow	Arrow, Inc.	390 Davis St.		Los Angeles	CA
	Mr. Dave	Elko	First Row Group	547 Jocom Way		Los Angeles	CA
	Mrs. Cindy	Blunden	Hot Lines, Inc.	#6 Hoover Pk		Palo Alto	CA
	Mr. Robert	Dorn	Valley Services	33 Cambridge P		Palo Alto	CA
	Mrs. Roxie	Jacobsen	Alpha Pic	174 Bellevue A		Palo Alto	CA

Now you have an easy way to locate any person in this database without having to bother with the **Find/Select** dialog.

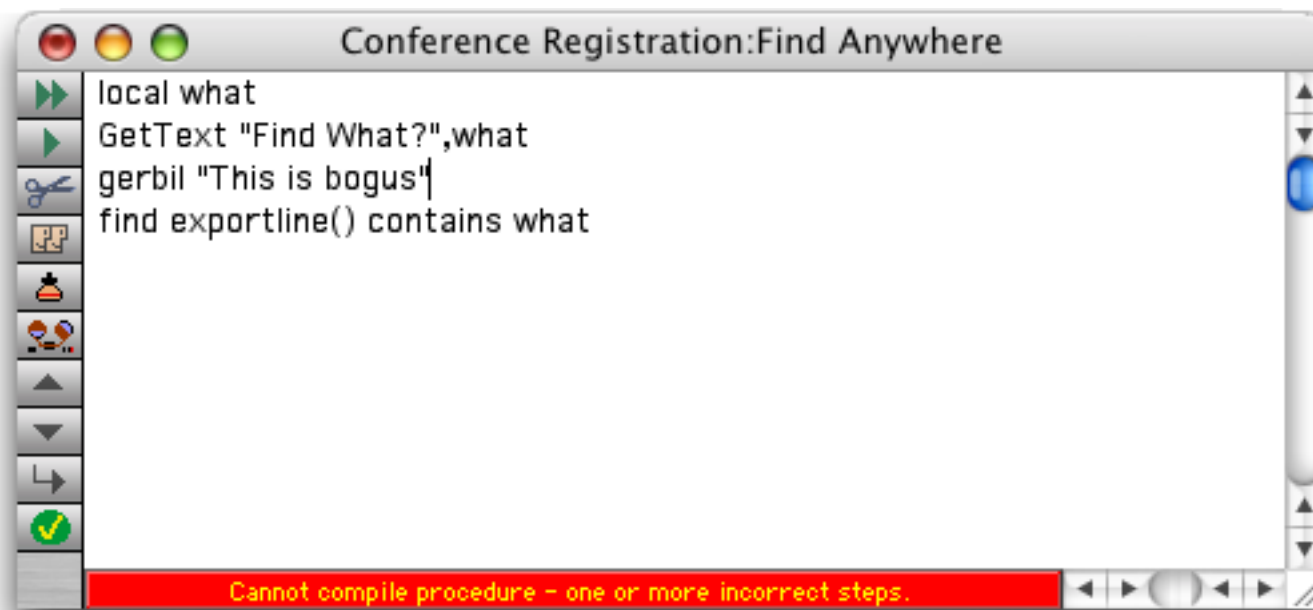
Checking for Mistakes

If you write a procedure yourself without using the recorder, you may make a mistake. You might misspell a keyword, forget to include a parameter, leave off a closing parenthesis, etc. Panorama will not let you use a procedure until you find and fix all of these mistakes.

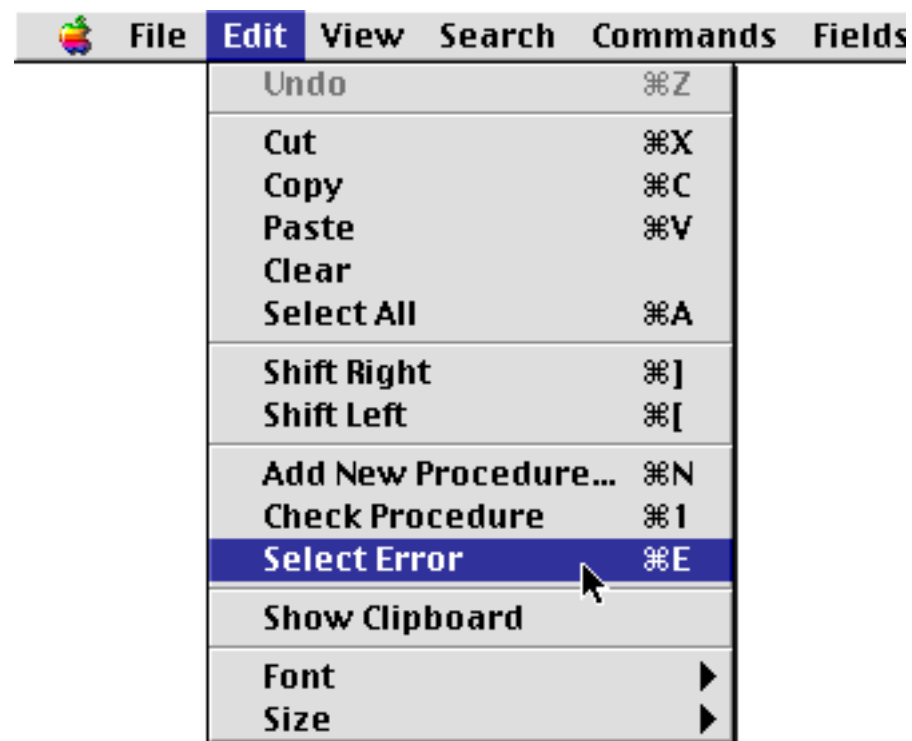
To help you find these mistakes Panorama provides the **Check Procedure** tool or menu command (Edit menu).



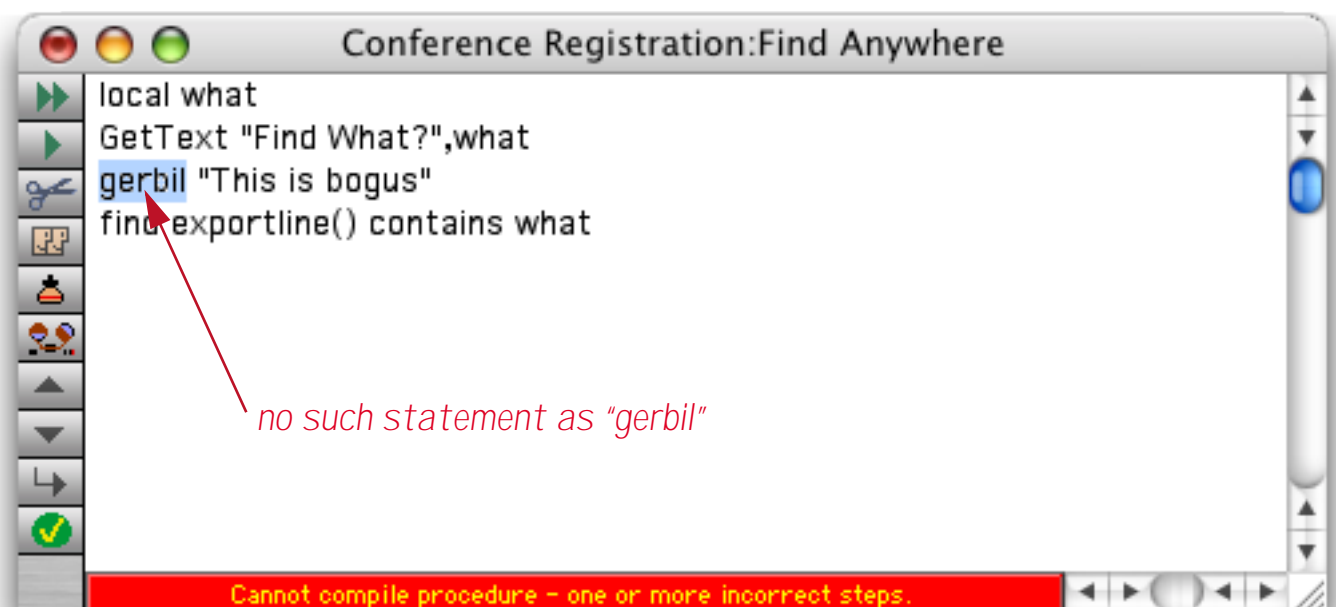
This tool scans the procedure looking for misspelled keywords and other mistakes. If it finds an error the status bar turns red and contains a description of the error.



Choose **Select Error** from the Edit menu if you would like Panorama to attempt to highlight the location of the error for you.



Panorama will identify the spot where it thinks the error occurred (see “[Mysterious Errors](#)” on page 222).



Correct the error, then use the **Check Procedure** tool again to see if there are any more mistakes. Repeat this process until the **Check Procedure** tool no longer finds any mistakes in your procedure. (Note: Panorama also automatically checks your procedure whenever you click on another window, save the database, switch to another view, or close the window containing the procedure.)

Mysterious Errors

Usually the **Check Procedure** tool in combination with **Select Error** is able to pinpoint the exact spot where the mistake in your procedure is located. Some types of mistakes, however, are not detectable right away, so Panorama actually will highlight the wrong spot in the procedure. Usually this is caused by a missing **endif** statement, mismatched parentheses, or mismatched quotes. If Panorama tells you that there is an error but the spot highlighted looks ok to you, check carefully above the spot where the error was flagged. The actual error may be many lines above the spot Panorama has flagged. If you still can't find the error, try splitting the procedure into several smaller procedures and checking each piece separately until you find the section containing the error.

Closing the Window When a Procedure is Finished

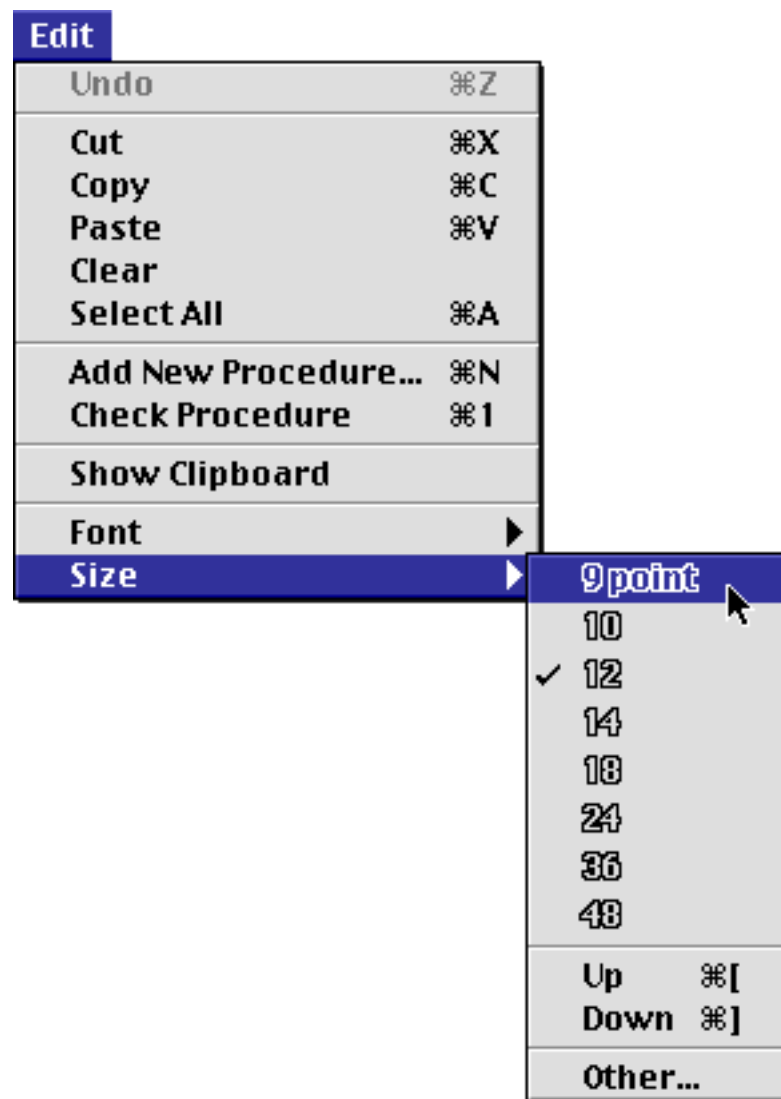
When you have finished writing a procedure you'll probably want to close the window for that procedure. If it is in a separate window, just click on the close box. If it is the only window for the database use the **View** Menu to flip to another view. You don't have to close the procedure window to use the procedure, but when you are sure it is working properly you will probably want to close it just to cut down on window clutter.

Re-Opening a Procedure

You can change any procedure at any time. Simply open the procedure with the **View** Menu, then make the changes. Don't forget that you can hold down the **Control** key (Macintosh) or **Alt** key (Windows) to make the procedure open into its own separate window.

Font and Size

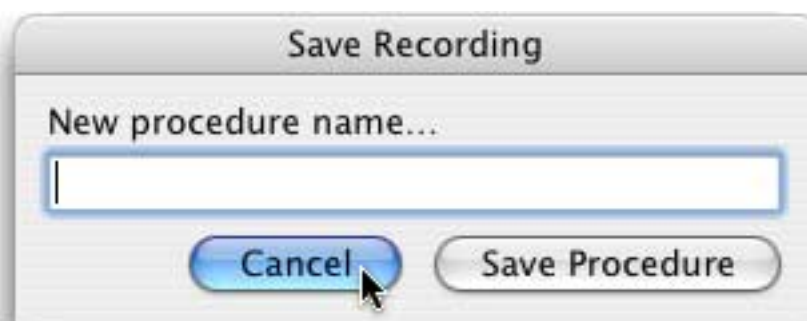
The **Font** and **Size** submenus (Edit Menu) allow you to change font and size of the procedures in this database. The font and size are always the same for every procedure in the database—you cannot set different procedures in the same database to different fonts or sizes.)



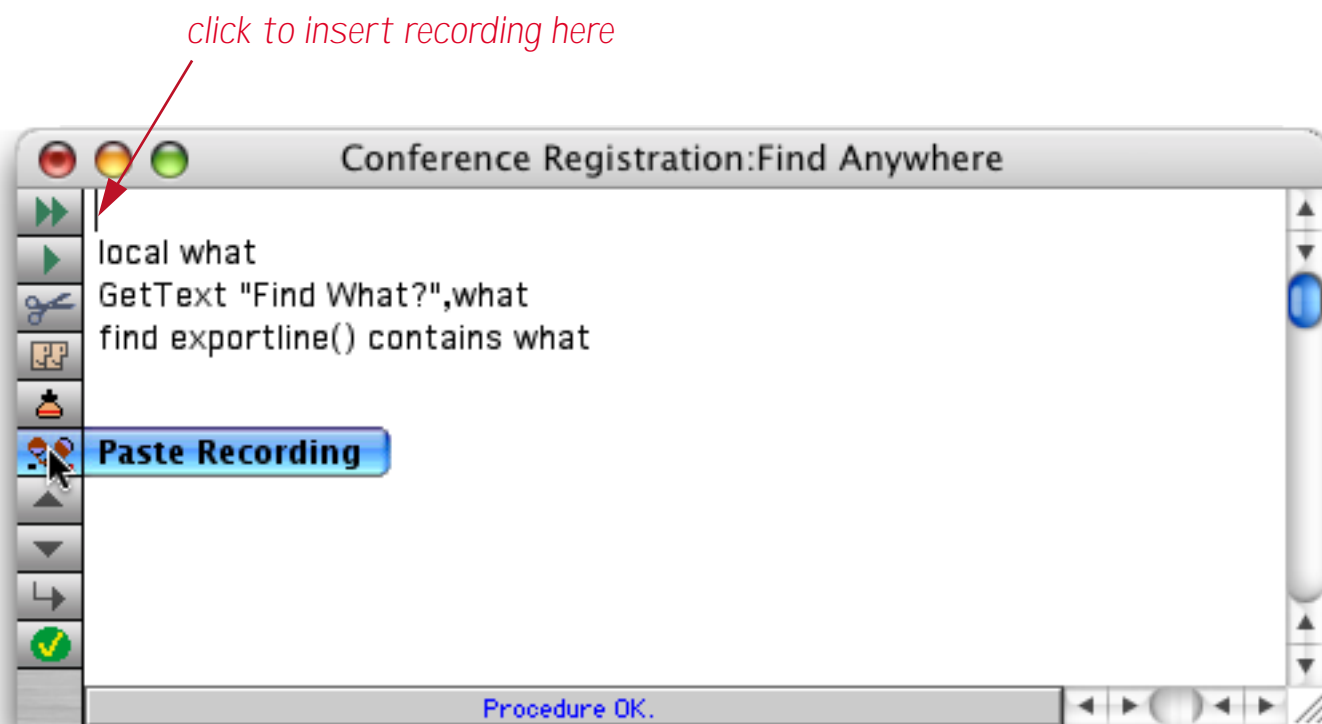
If you are using a Windows system we recommend that you stick with one of the four fonts installed with Panorama: Alpine, Block, City or Yankee (the default is Alpine 12). These fonts are designed to be able to display some of the special characters used by Panorama that are not normally available on Windows systems (\neq , \leq , \geq etc.)

Adding a Recording to an Existing Procedure

Earlier you learned how to create a new procedure by recording (see “[Creating a Procedure with the Recorder](#)” on page 212). It’s also possible to use the recorder to add statements to an existing procedure. To do this, start the recorder normally, and record the steps you want to include. When the steps have been completed click on the recorder again to stop the recording. When Panorama asks you what name you want to give the new procedure, click the **Cancel** button.



Now go to your procedure window and click on the spot where you want the recording to be inserted. Then choose the **Paste Recording** tool.



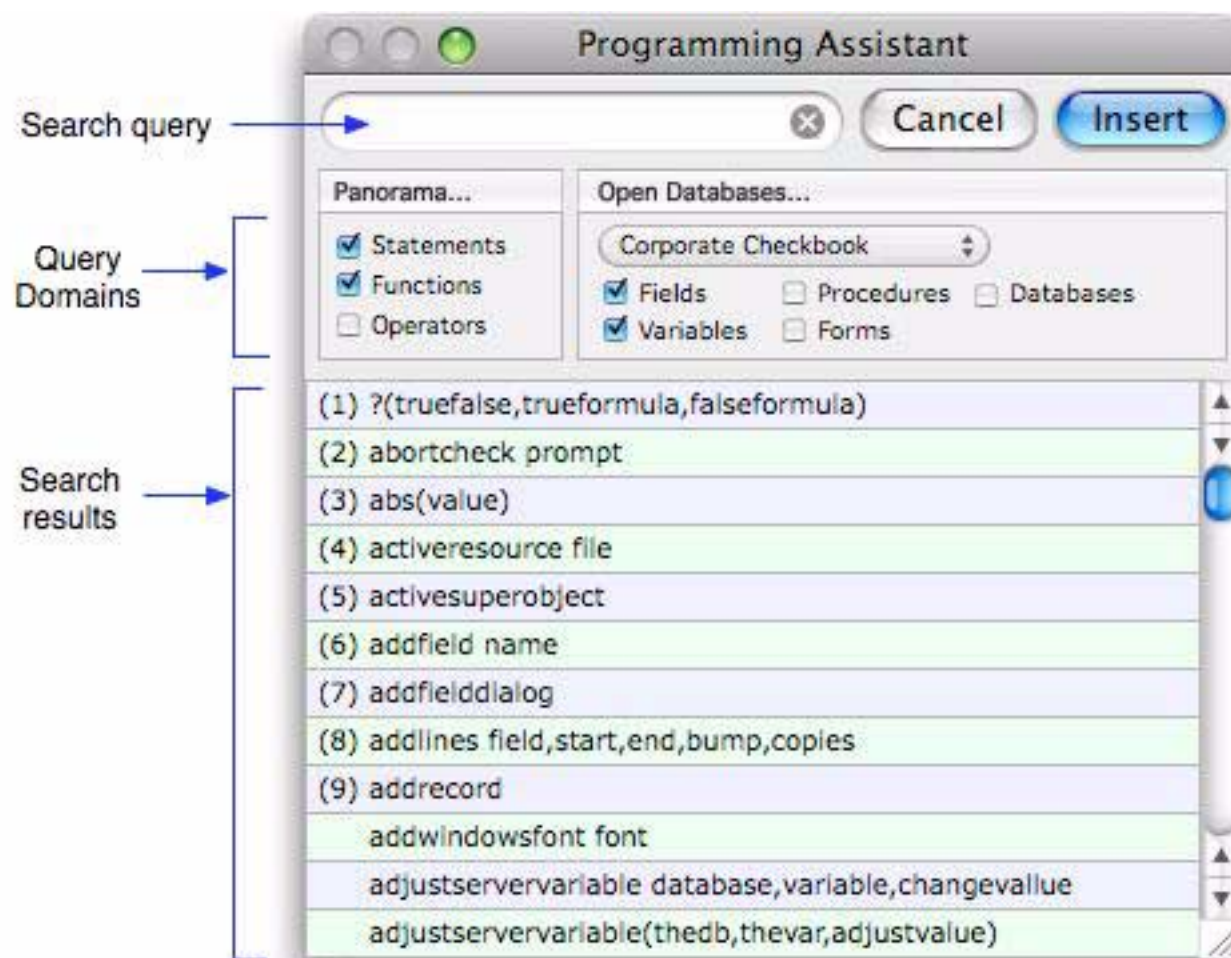
Panorama will insert the recording into the procedure. If necessary you can edit the recorded statements or use them "as is."

Programming Helpers

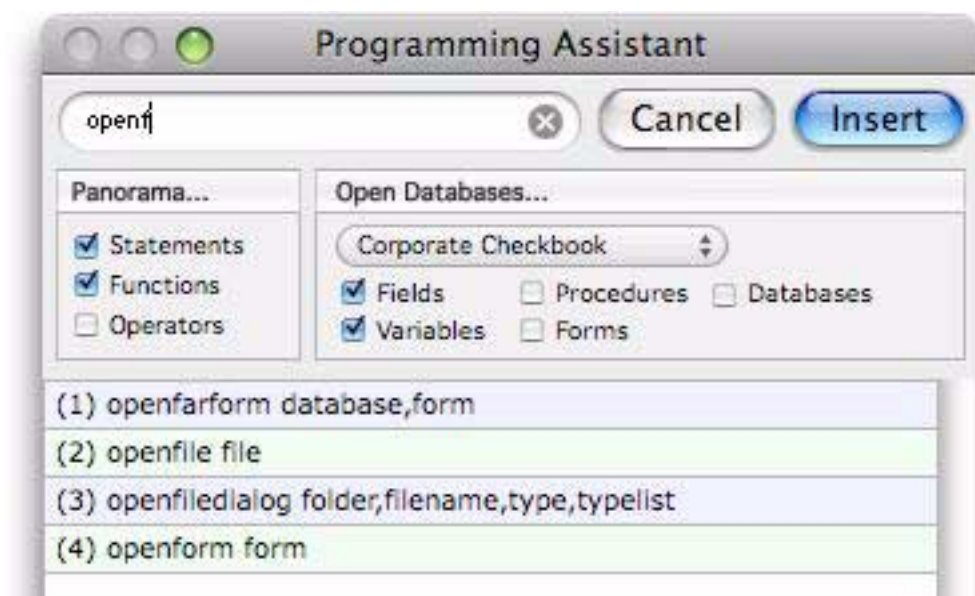
There are a wide variety of different ingredients that can go into writing a Panorama program — statements, functions, fields, variables, you name it. It's a lot to learn and keep track of, and even the top experts here at ProVUE Development can't keep all of it memorized. Fortunately, that's not necessary. Panorama has a number of different aids to help you find all the right ingredients to complete your programming tasks, including the **Programming Assistant** dialog, the **Programming Context Menu**, and the **Programming Reference** wizard.

The Programming Assistant Dialog

The **Programming Assistant** dialog makes it easy to type in any statement, function, field, variable or other programming element. You only need to type in the first few characters and the assistant will fill in the rest. Within a procedure you can open the assistant from the **Edit** menu, or by pressing **Command-Quote**, or by right clicking on the procedure and choosing **Programming Assistant** from the **Help** submenu. No matter how you open it, the dialog opens and displays a list of items that can be inserted into the procedure.



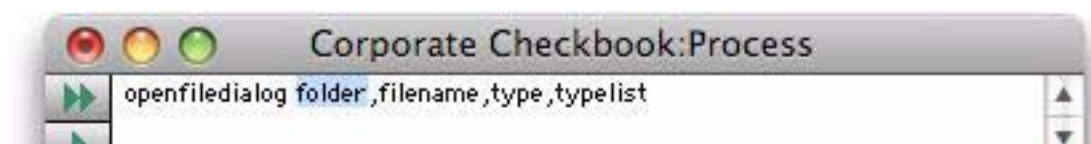
Initially the list may contain a couple of thousand items. To narrow this down, type in part of the item you are looking for. For example if you want to open a file, start by typing in **openf**.



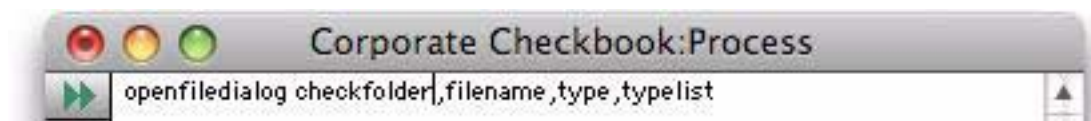
Once the item you want is visible, you have three choices for inserting it into the procedure:

- 1) If the item is one of the first nine, press **[1]** to **[9]**.
- 2) Click once on the item, then press the **Insert** button.
- 3) Double click on the item.

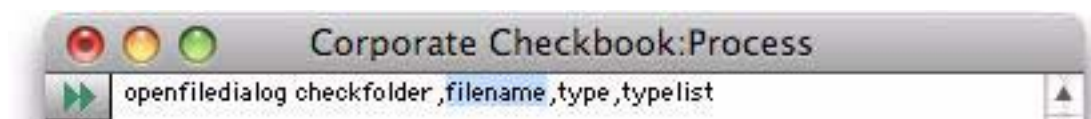
If the item being inserted is a statement or function that has parameters, the first parameter will automatically be selected.



You can just start typing to enter the first parameter.



When you're ready to skip to the next parameter, choose **Select Next Parameter** from the **Edit** menu (or simply press **Command-Comma**).



You can continue the process until all of the parameters are completed.

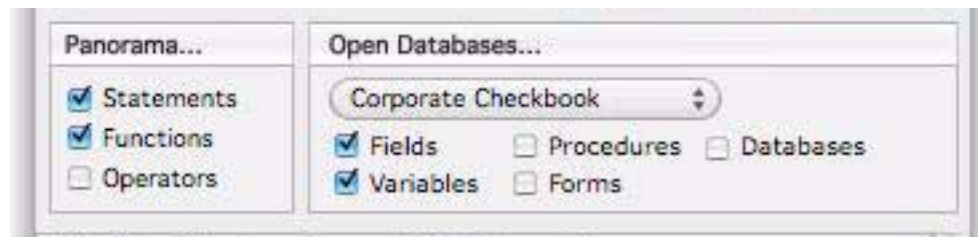
Using the Assistant from the Keyboard

The **Programming Assistant** is designed so that you can use it entirely from the keyboard, without touching the mouse.

- 1) Press **Command-Quote** to open the dialog.
- 2) Type in enough characters so that the item you want to insert is one of the first nine items.
- 3) Press **[1]** to **[9]** to insert the item into the procedure.
- 4) (Optional) Use **Command-Comma** to skip thru each parameter

Assistance Domains

The **Programming Assistant** has eight different domains that it can assist with. To reduce confusion, you can choose to have only some of these domains displayed. Simply check the domains you want to see.



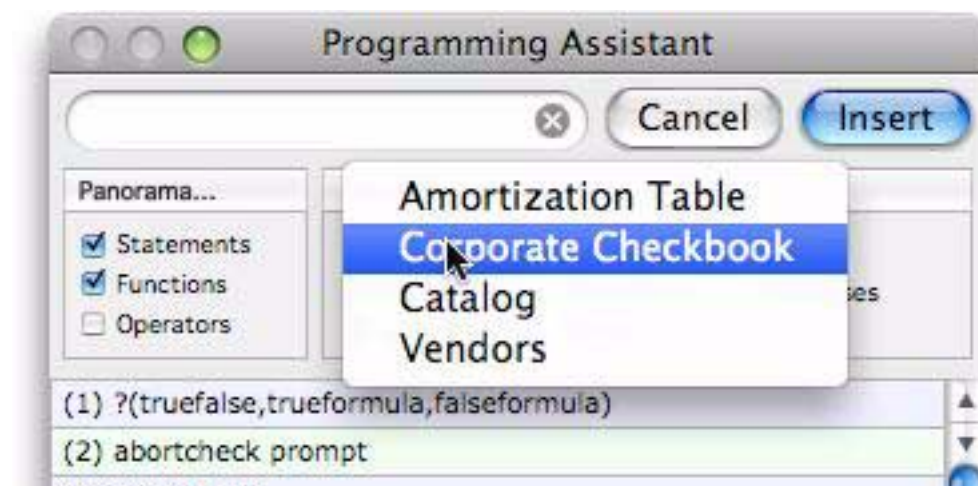
Three of these domains (*Statements*, *Functions* and *Operators*) are universal. These domains are the same no matter what database is active.

Domain	Description
Statements	Programming statements that can be used in a procedure.
Functions	Calculation functions that can be used in a formula (math, text, dates, etc.)
Operators	Operators that can be used in a formula (+, /, *, etc.)

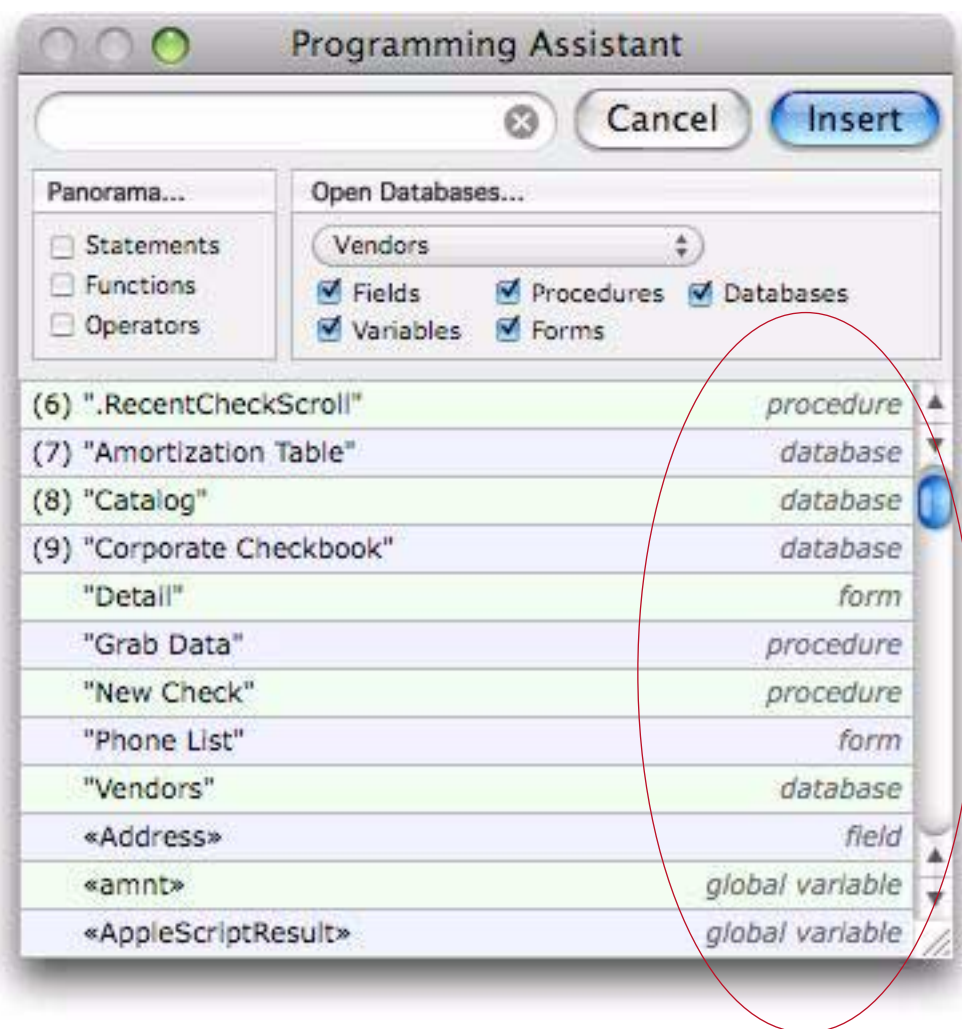
The other five domains change depending on what database is selected.

Domain	Description
Fields	Field names in the current database.
Variables	Global and fileglobal variables. Local variables are not listed, and variables are not listed until the procedure defining them has been run.
Procedures	Names of procedures in the current database (useful for setting up the call statement, for example)
Forms	Names of forms in the current database
Databases	Names of all open databases

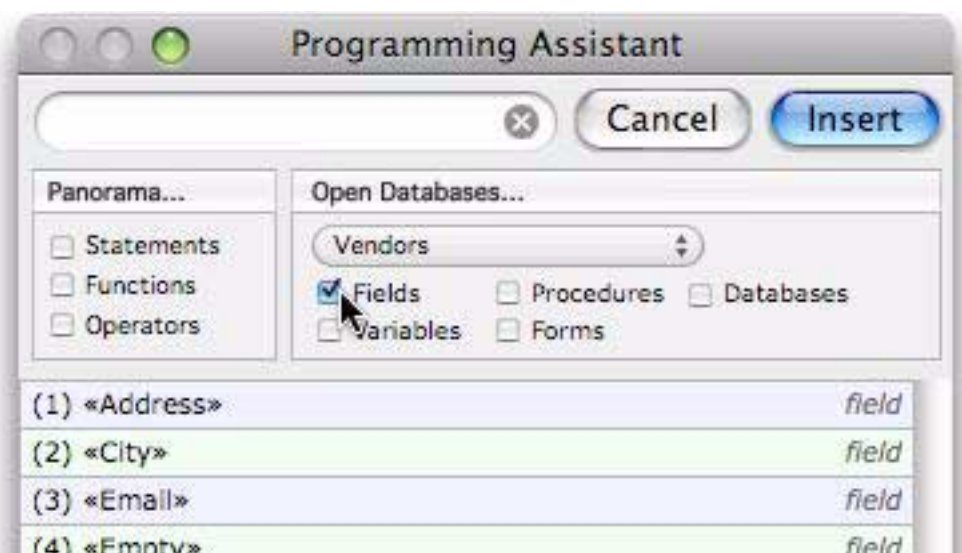
When the dialog first opens these domains display items from the current database. Use the pop-up menu to change what database is listed.



When the assistant is displaying database specific information, it displays the type of each item on the right (procedure, database, form, field, variable, etc.)

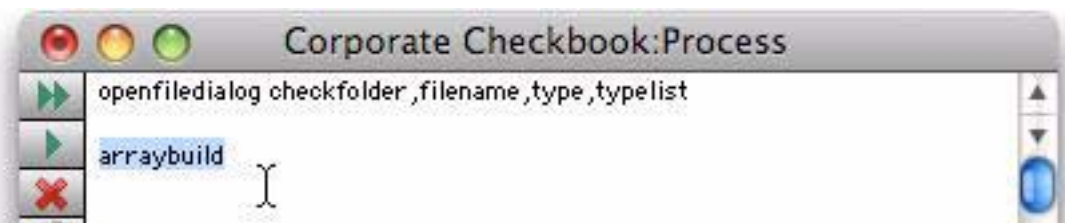


If you want to display only a single domain, just right click on the domain or hold down the option key when you click on the domain. This automatically unselects all of the other domains and selects only the domain you clicked on. In this example I right clicked on the **Fields** checkbox, so now only fields in the Vendors database are listed.

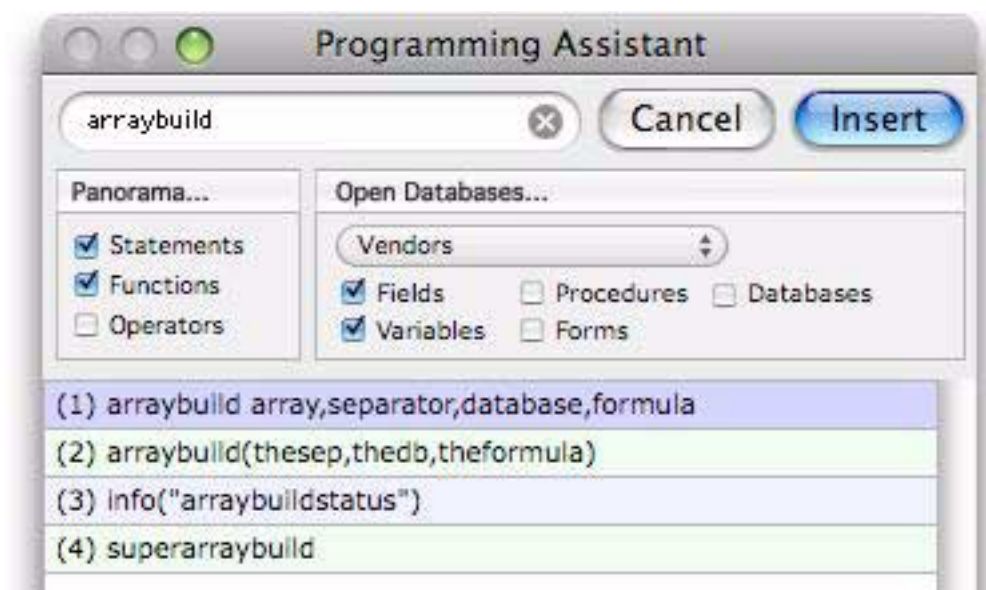


Getting Assistance with a Selection

If text is already selected when you open the dialog, the assistant will automatically pre-select items that match the selected text. For example, suppose you want to use the `arraybuild` statement but you don't remember what the parameters are. You can start by typing `arraybuild` into the procedure and selecting it.



Then press **Command-Quote** to open the assistant.



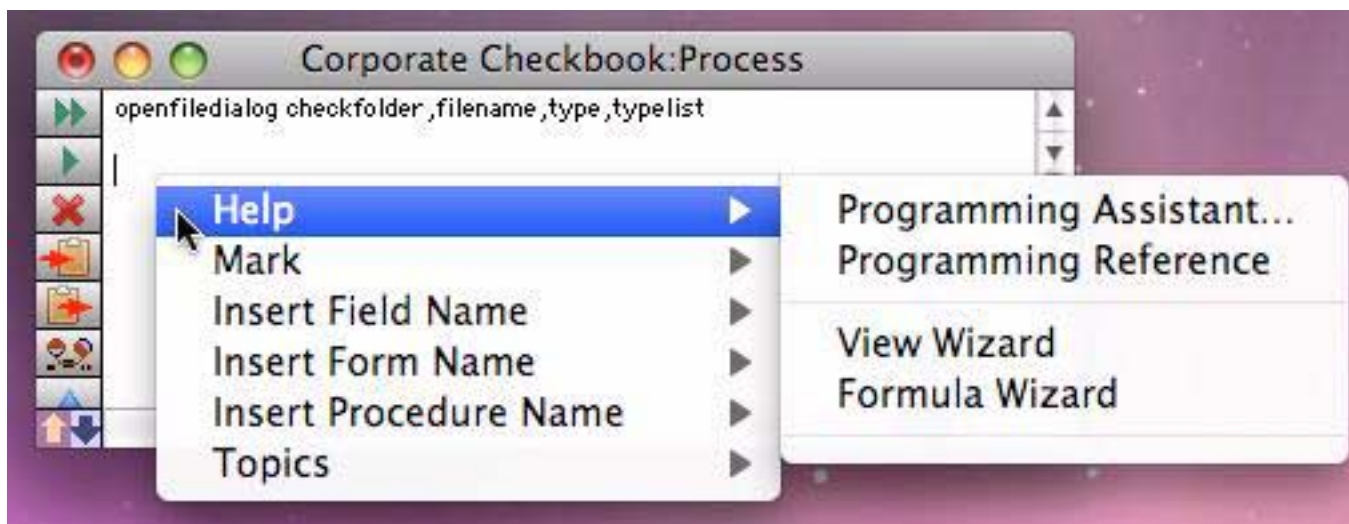
Voila — `arraybuild` is already selected.

Smart Text Insertion

The Programming Assistant tries to be smart about inserting text. For example if you insert a field or variable name that contains unusual punctuation, it will automatically add the necessary chevrons (« and »). However, it won't do that if you are inserting the name into quotes or if the selection is already surrounded by chevrons, because in those cases it wouldn't be correct. For the most part you probably won't notice this as it usually does the right thing.

The Programming Context Menu

When you're editing a procedure you can right click in the editing window to activate a special menu that is chock full of programming goodies. The exact contents of the menu will vary depending on the database and the current selection, but it will generally look something like this:

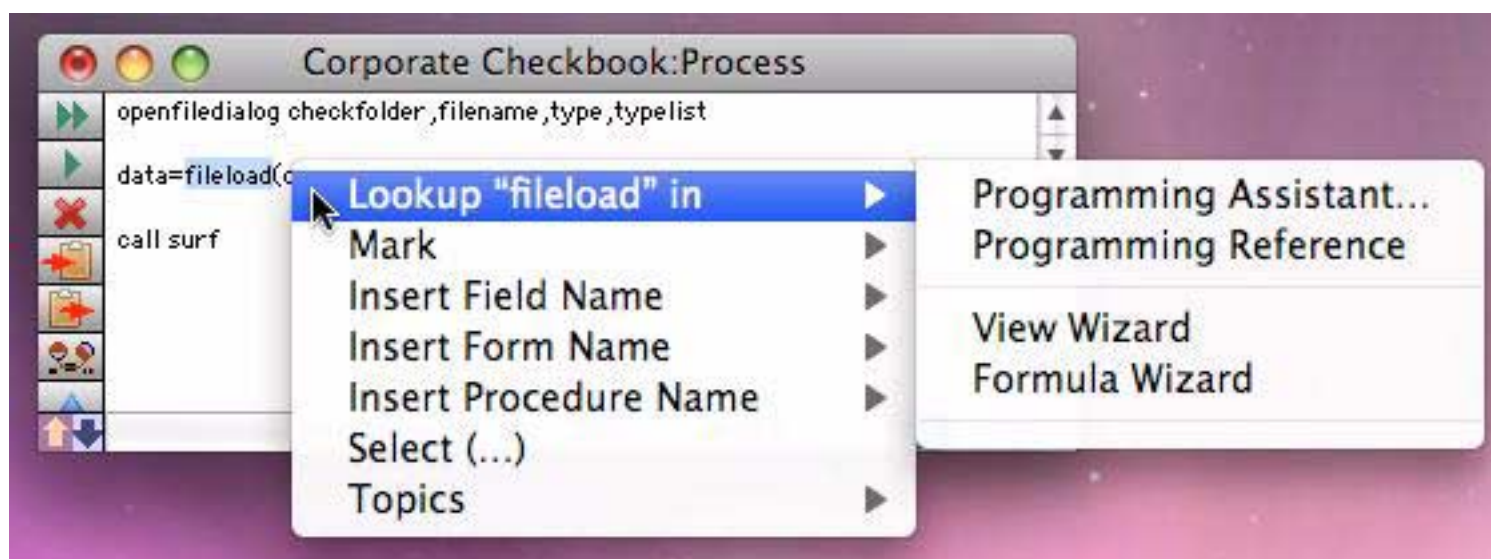


This menu has six sub-menus, which are described in the following sections.

Help Submenu

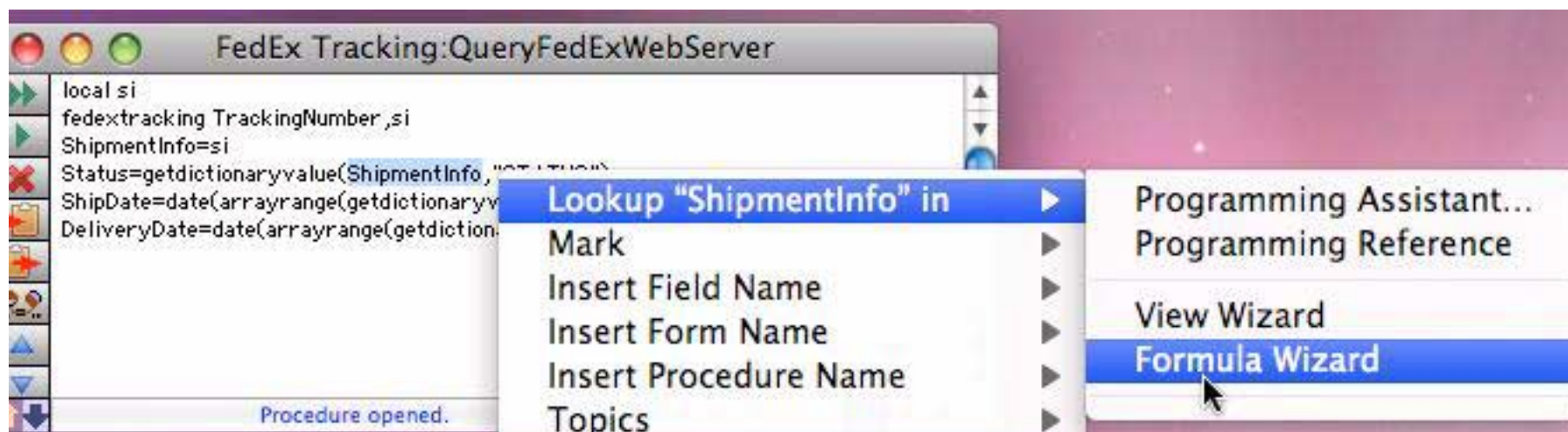
This menu allows you to access various forms of programming help. Simply select the type of help you need at the moment (see "[The Programming Assistant Dialog](#)" on page 225, "[Programming Reference Wizard](#)" on page 237, "[Using the View Wizard with Procedures](#)" on page 344 and "[Using the Formula Wizard](#)" on page 29).

If there is any text selected, the Help Submenu changes into the Lookup submenu, like this.

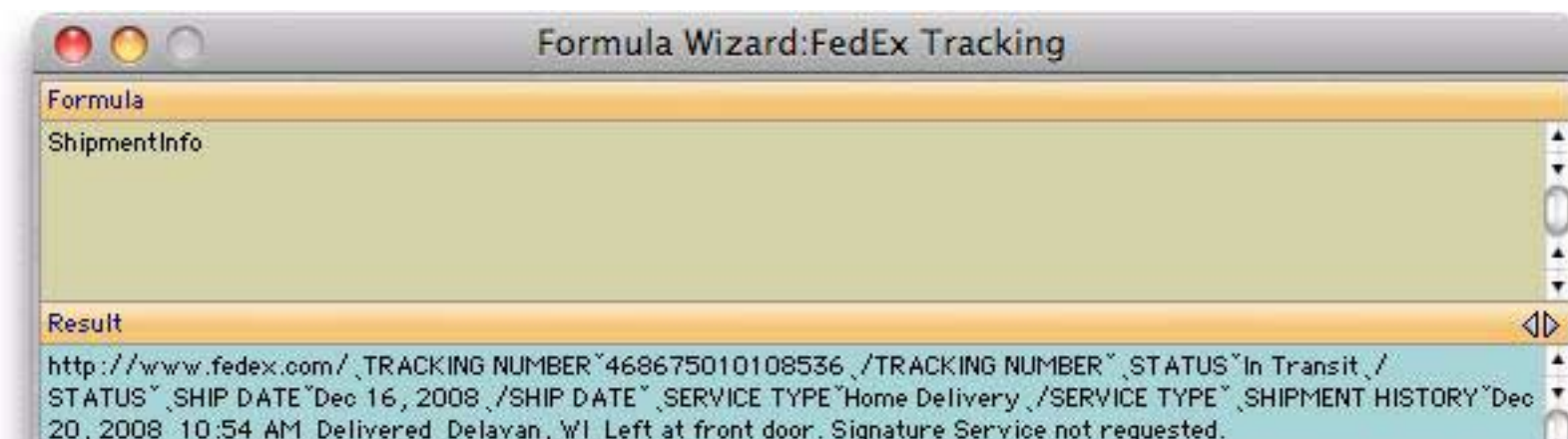


In this situation selecting an option in this submenu not only opens the window you requested, it also looks up the selected text. For example, if you select the word fileload (as shown above) and then choose **Programming Reference**, the Reference will open right to the page for the fileload(function.

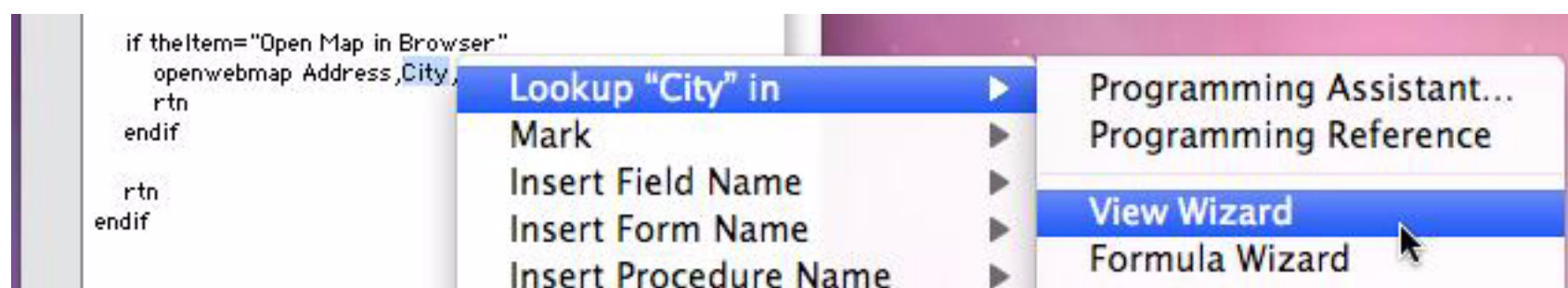
You can check the value of a field, variable or formula by selecting it and then open the **Formula Wizard** (Note: this does not work for local variables).



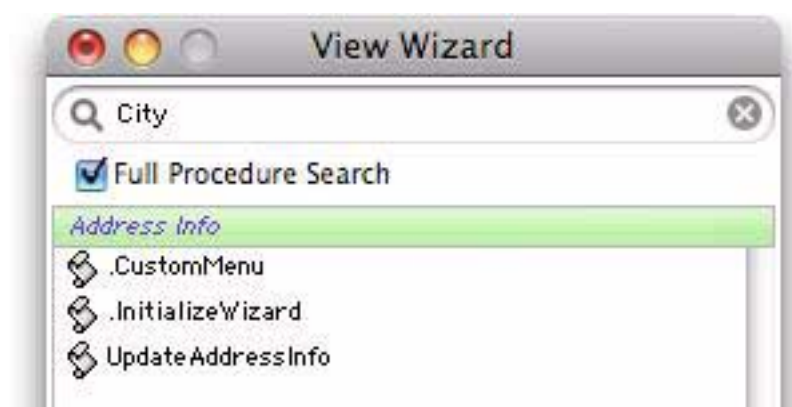
The **Formula Wizard** shows the value (if any) of the item you selected (see [“Using the View Wizard with Procedures”](#) on page 344).



To find out where else in the database a particular field, variable or procedure has been used, select the item and then open the **View Wizard**



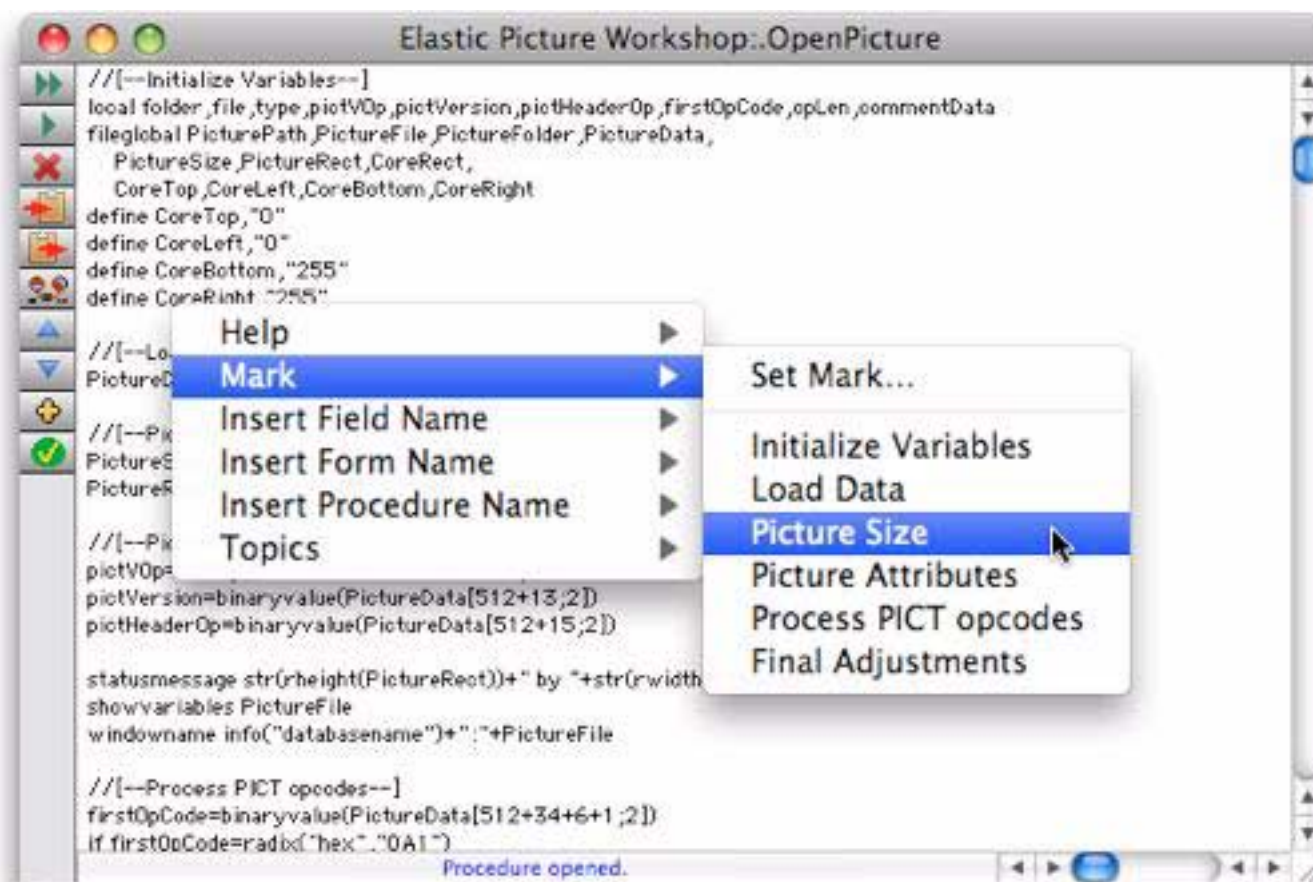
The **View Wizard** will show all of the procedures that contain the selected text.



To actually open any of the matching procedures and find the searched for item, double click on the procedure name. See [“Using the View Wizard with Procedures”](#) on page 344 for more information.

Mark Submenu

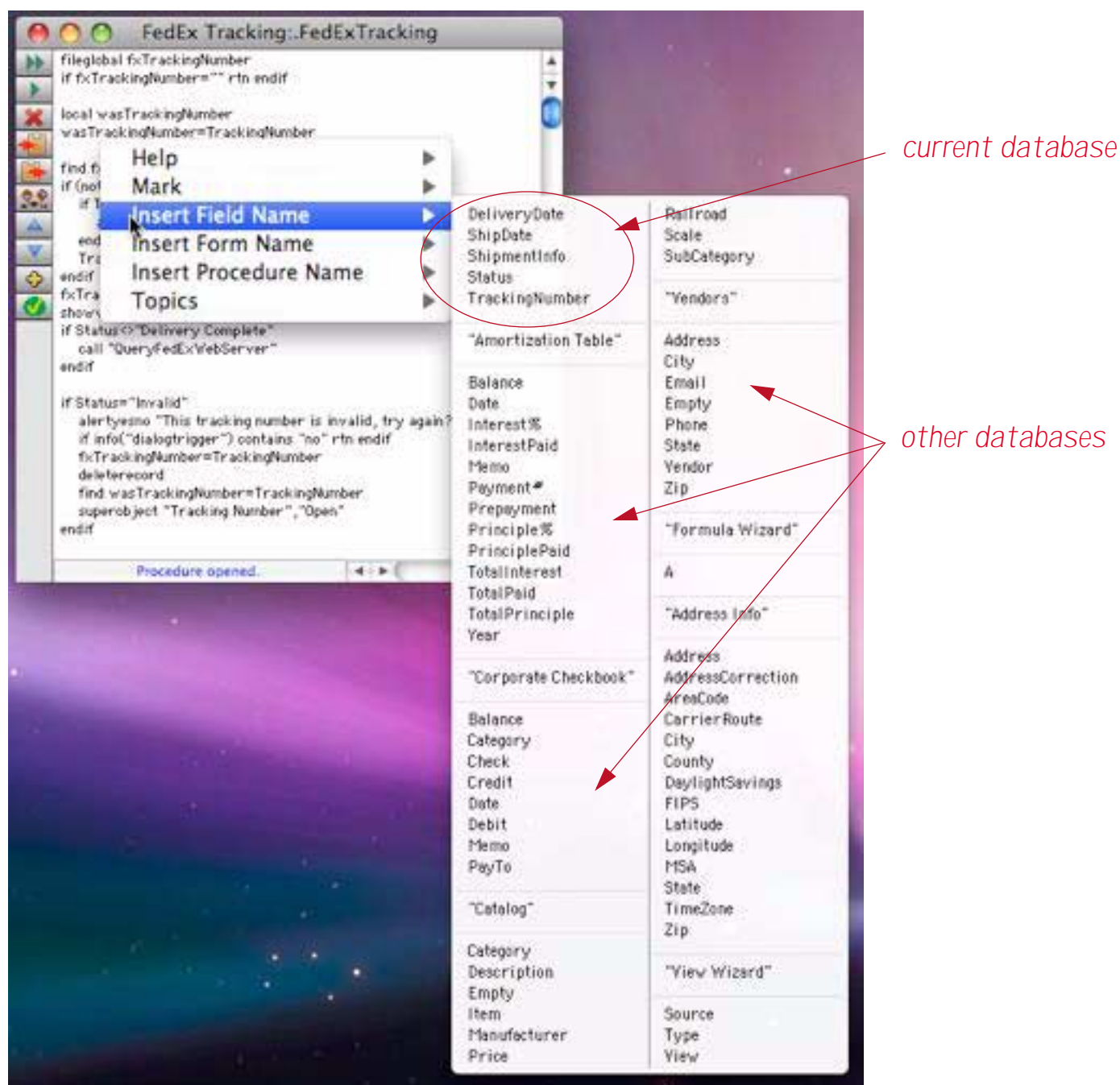
A single procedure may include up to 32,000 characters of text. A procedure that long would be more than 20 pages long if printed. As a procedure grows it can be difficult to navigate within the procedure itself. The **Mark** submenu allows you to create “bookmarks” within the procedure that you can quickly jump to.



This submenu is a duplicate of the Mark menu in the menu bar. To learn more about this menu, and how to create marks, see “[Organizing Large Procedures \(The Mark Menu\)](#)” on page 305.

Insert Field Name Submenu

The **Insert Field Name** submenu lists all of the fields in all open databases. The fields in the current database are listed first, then all of the other databases. To insert a field name into the procedure simply select it from this submenu.



The field names are normally listed alphabetically within each database. If you would prefer to list them in the order that they appear in the database, hold down the **Option** key when you bring up the menu.

When a fieldname is inserted into the procedure Panorama will automatically adjust it as necessary for the situation. For example when a multi-word field name or a field name with punctuation is inserted, the necessary chevrons will be added automatically (for example «**Tracking Number**» or «**P/E Ratio**»). However, if the field name is being inserted in between quotes, the chevrons won't be added (for example "**Tracking Number**" or "**P/E Ratio**"). The quotes have to be put in first for this feature to work.

Insert Form Name Submenu

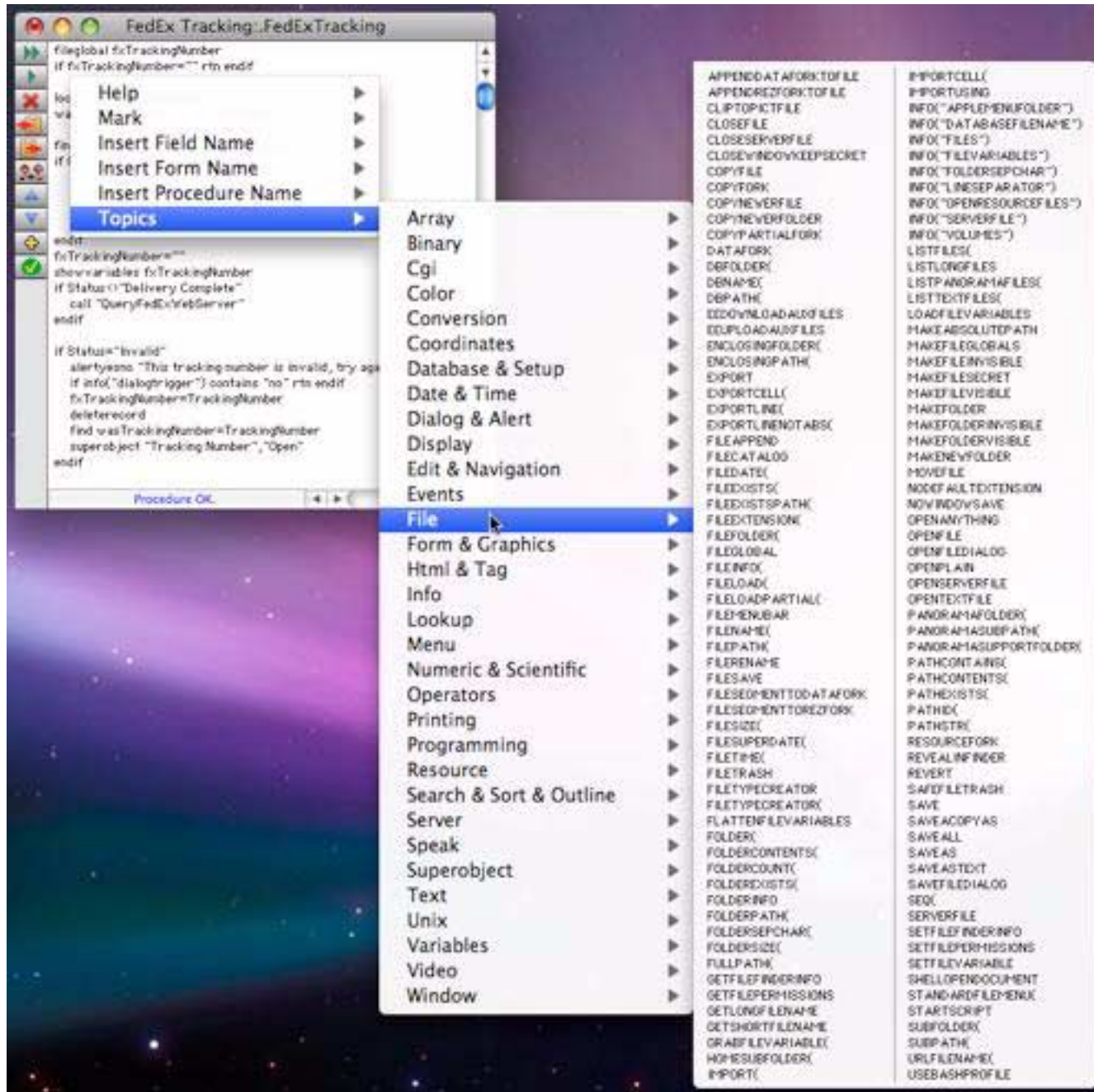
This submenu works just like the **Field Name** submenu, but lists forms instead of fields.

Insert Procedure Name Submenu

This submenu works just like the **Field Name** submenu, but lists forms instead of fields.

Topics Submenu

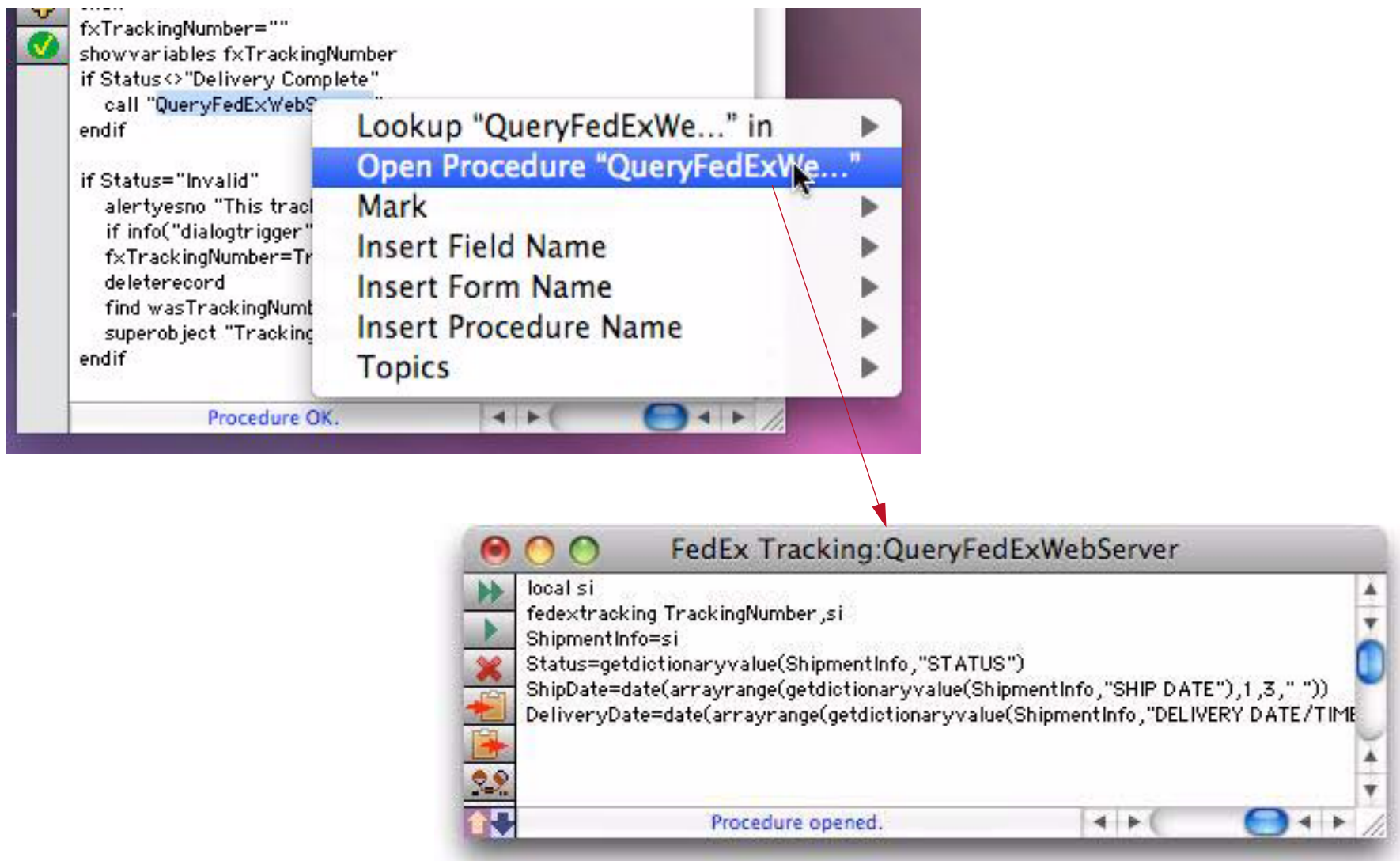
The **Topics** submenu lists over a thousand Panorama statements and functions, divided into about two dozen subtopics. (Some statements or functions may be displayed under more than one subtopic, and some less used statements or functions may not be listed at all. Use the **Programming Assistant** or **Programming Reference** for 100% coverage.)



To insert a statement or function just choose it from the menu.

Opening a Procedure or Form

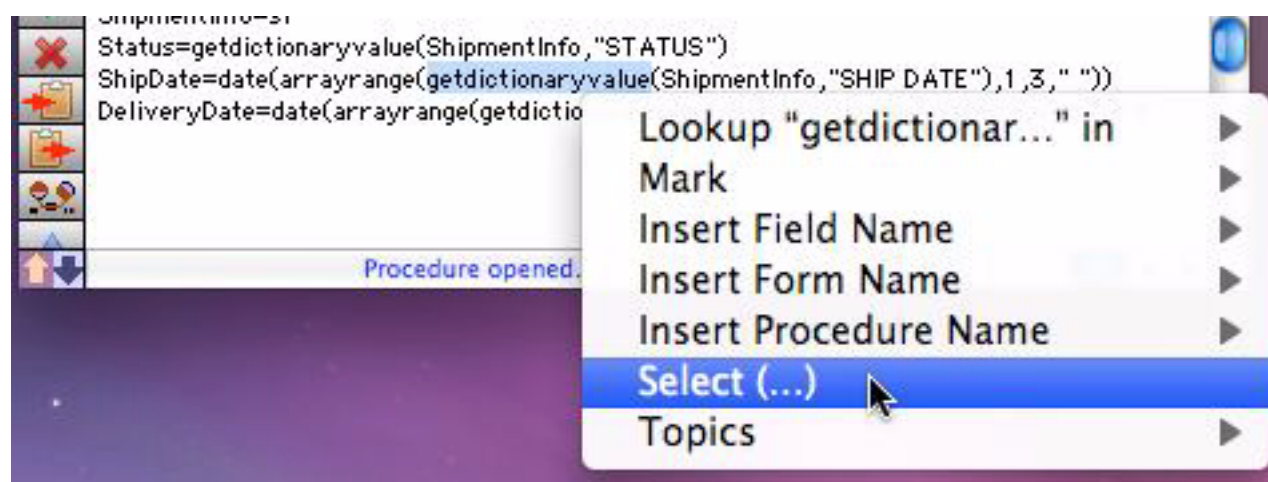
If Panorama notices that you've selected the name of a procedure or a form, the context menu will contain an option to directly open that procedure or form.



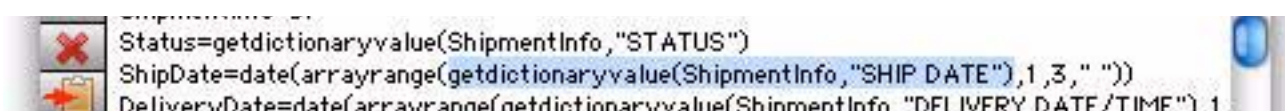
If you have both a procedure and a form with the selected name, options for opening both will be listed.

Selecting Parentheses Contents

If the current selection is just in front of a (symbol, choose **Select (...)** to automatically select all of the text enclosed in the parentheses.



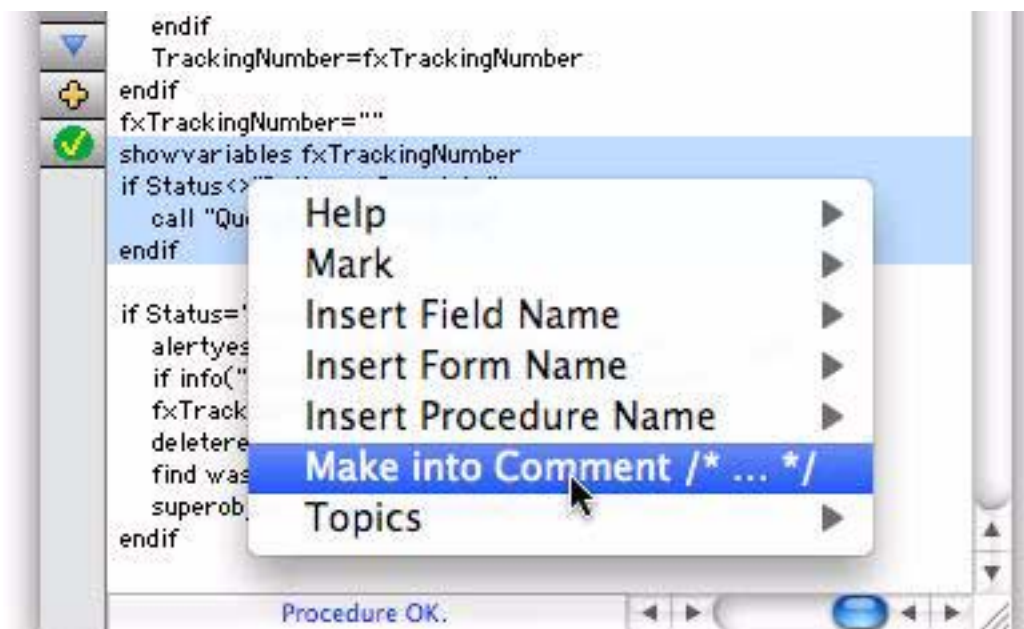
When this menu item is chosen the selection is extended up to the matching right parenthesis.



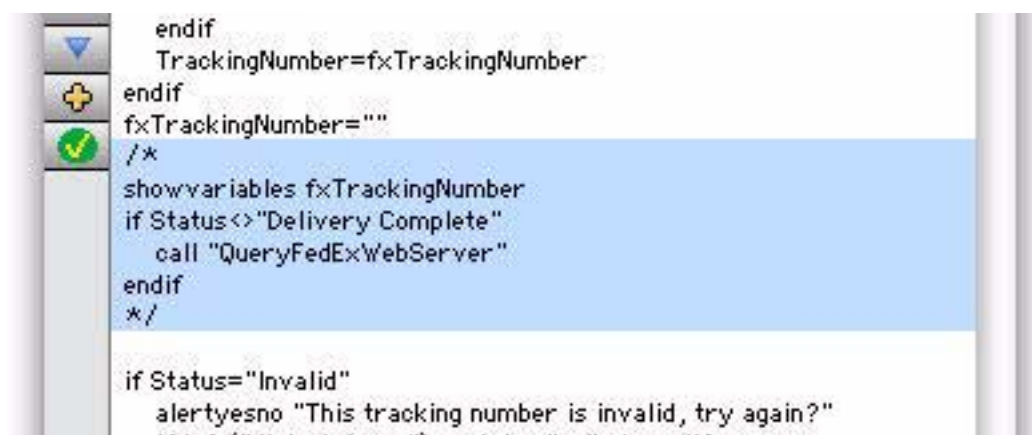
This feature will work with nested parentheses, however, it will be confused by text constants that contain parentheses (for example "(" or ") ").

Comment/Uncomment

If you've selected multiple lines of text, you can quickly "comment them out" using the context menu.



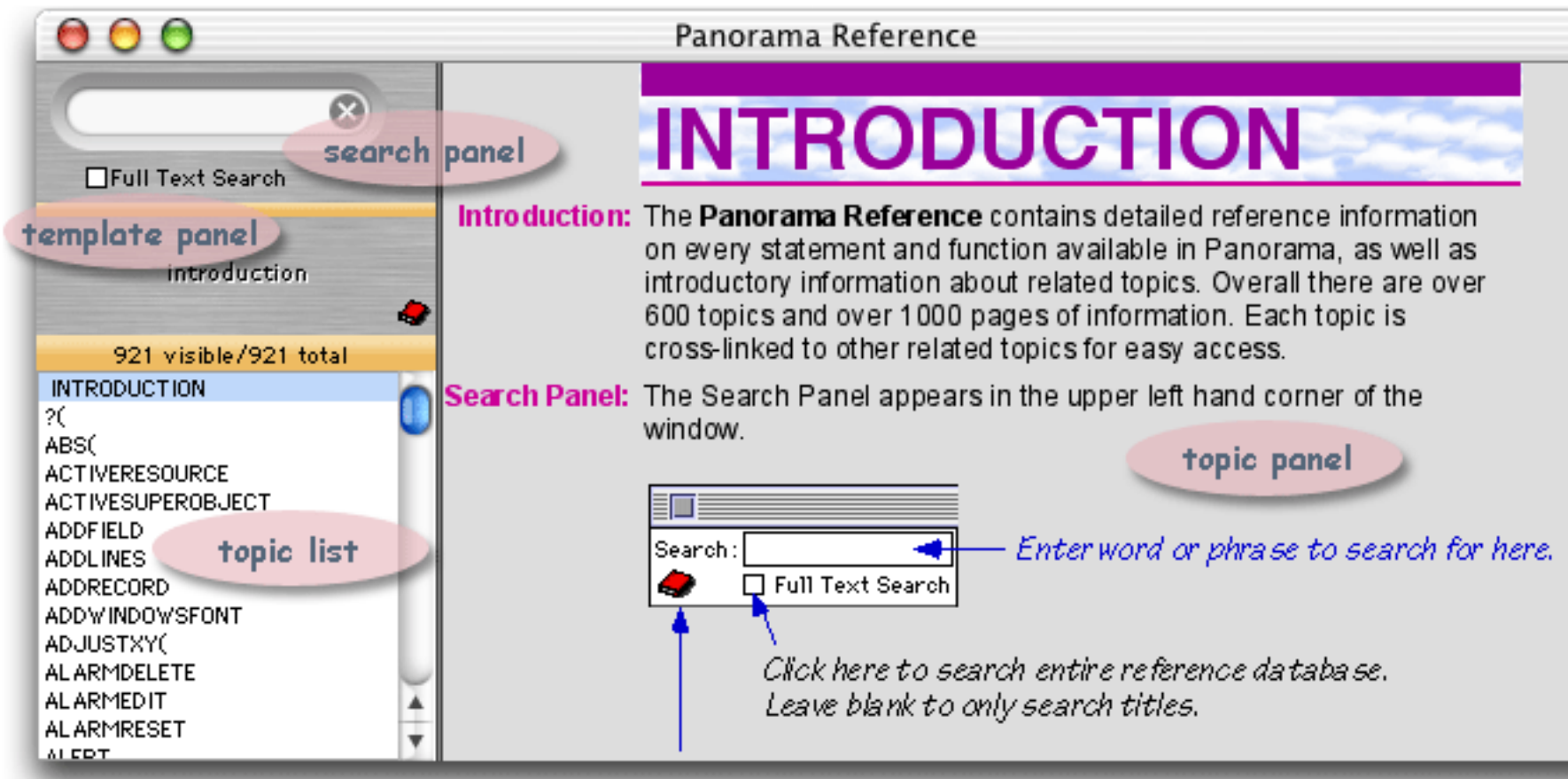
The "commented out" section of code will be skipped when this procedure is run. This can be useful for testing.



For more information on this technique see "[Notes To Yourself](#)" on page 304.

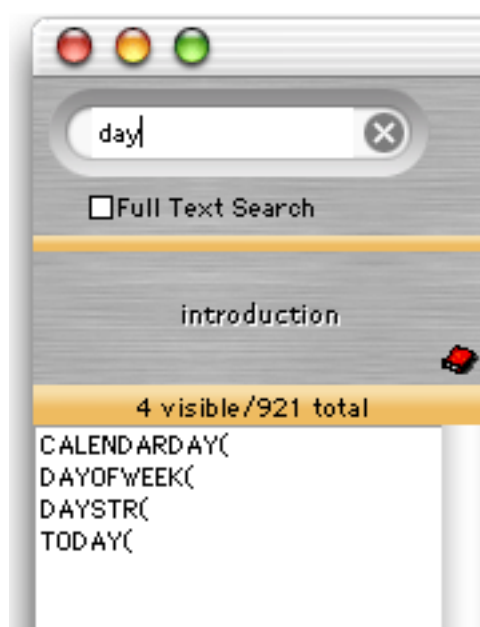
Programming Reference Wizard

Panorama has over one thousand functions and statements that you can use in formulas and procedures. Most of these have one or more parameters that must be used correctly. Even here at ProVUE Development we can't keep all of this memorized. To help, we created the **Programming Reference** wizard, an instant, on-line reference to what's what in Panorama formulas and programs. You can open this wizard from the Documentation submenu of the Wizard menu, or by pressing **Control-R** (Macintosh only). Once the wizard opens the reference window is divided into four sections: search panel, template panel, topic list and topic panel.

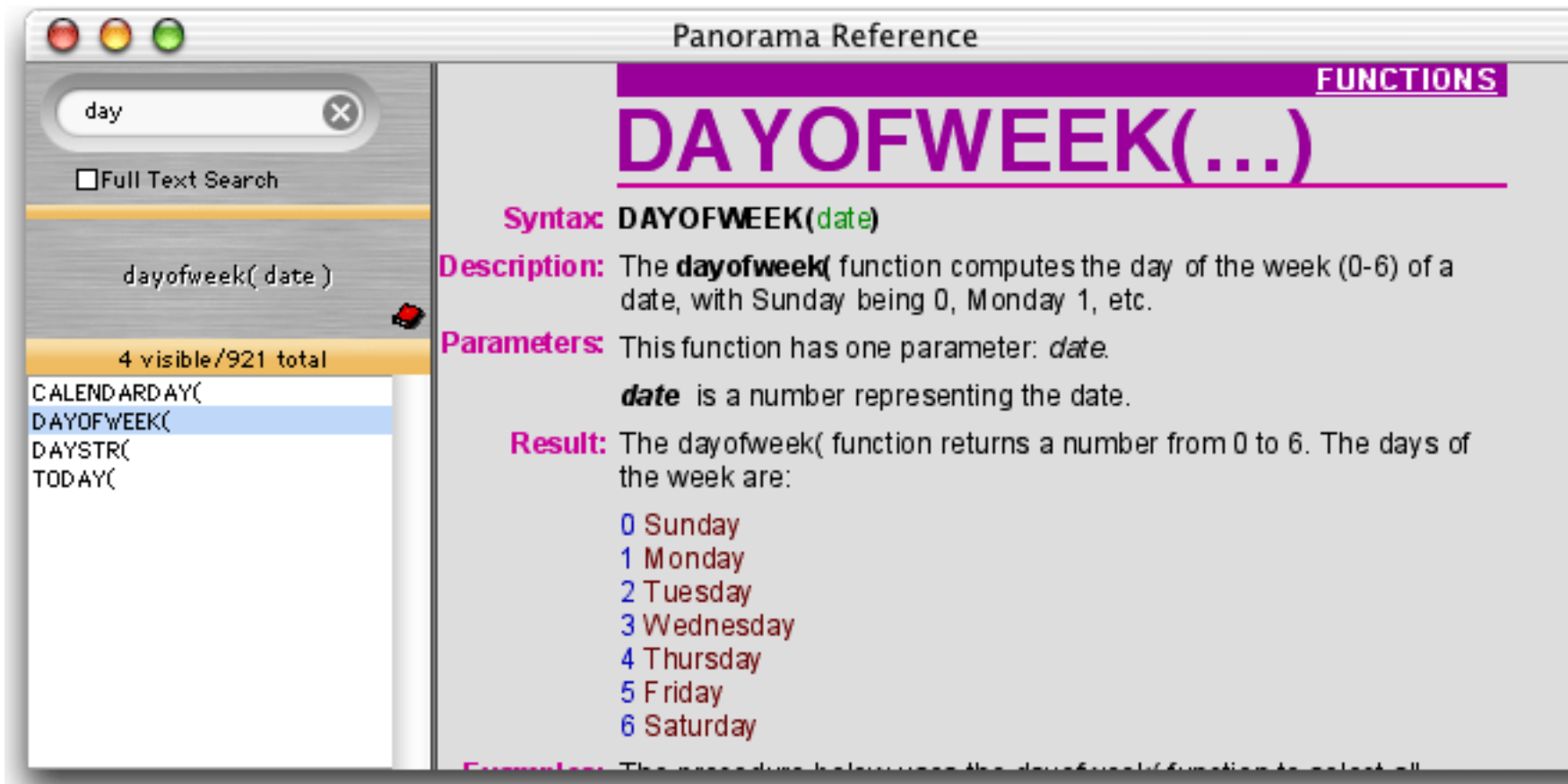


Navigation Using the Search Panel and Topic List

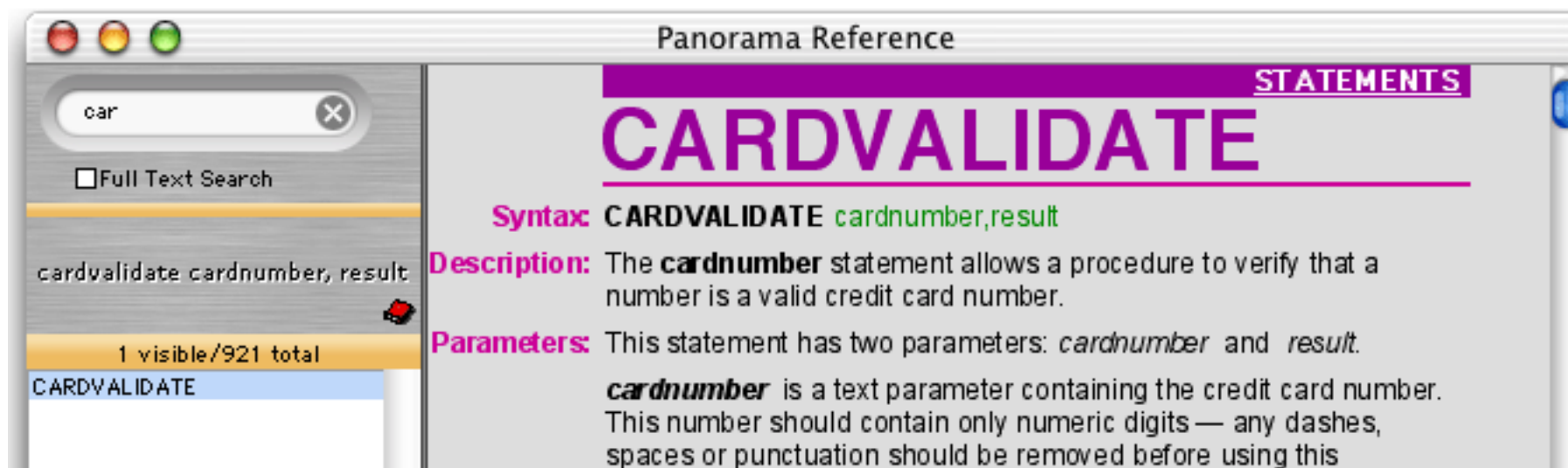
The search panel and topic list work together to help you locate a specific topic. As you type into the search panel, the topic list updates to show topics that match.



When you see the topic you want, click on it to display the topic in the topic panel.



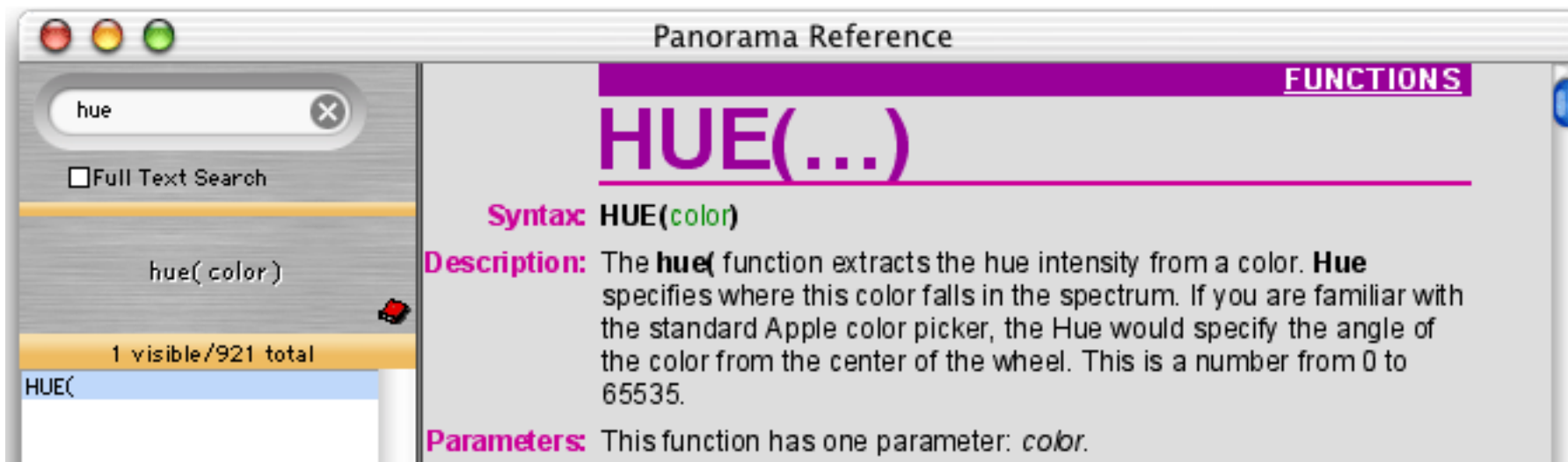
If there is only one topic in the topic list, the wizard will display the topic automatically (without having to click on the list).



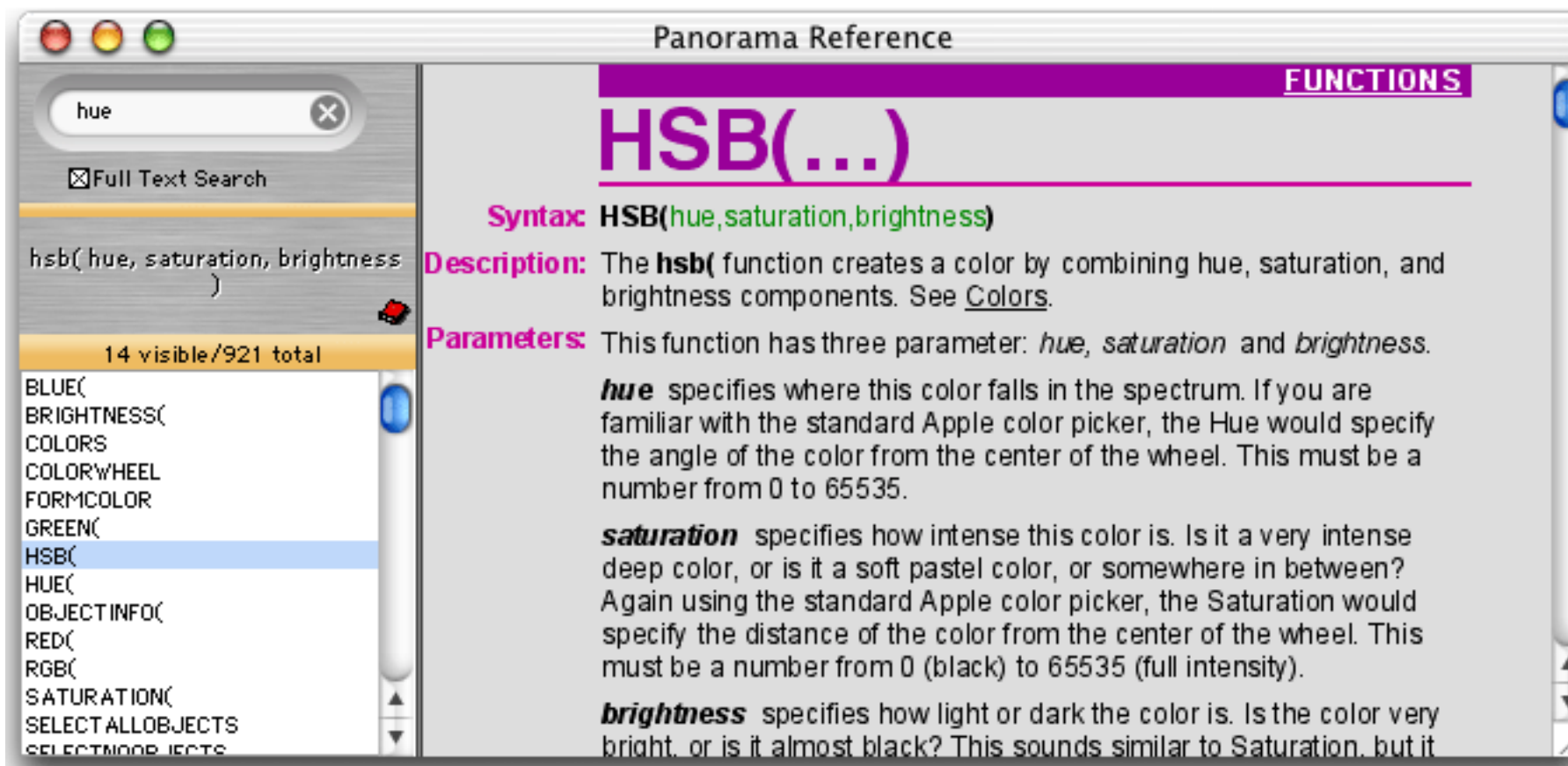
To quickly erase the query in the search panel, click on the  button.

The Full Text Search Option

The search panel normally searches only the name and category of each topic. When the **Full Text Search** option is checked the wizard will also search the complete text of each topic. This makes it possible to quickly find every topic that references a particular function or statement, as well as the function or statement itself. For example, a normal search for the word **hue** will turn up only one match.

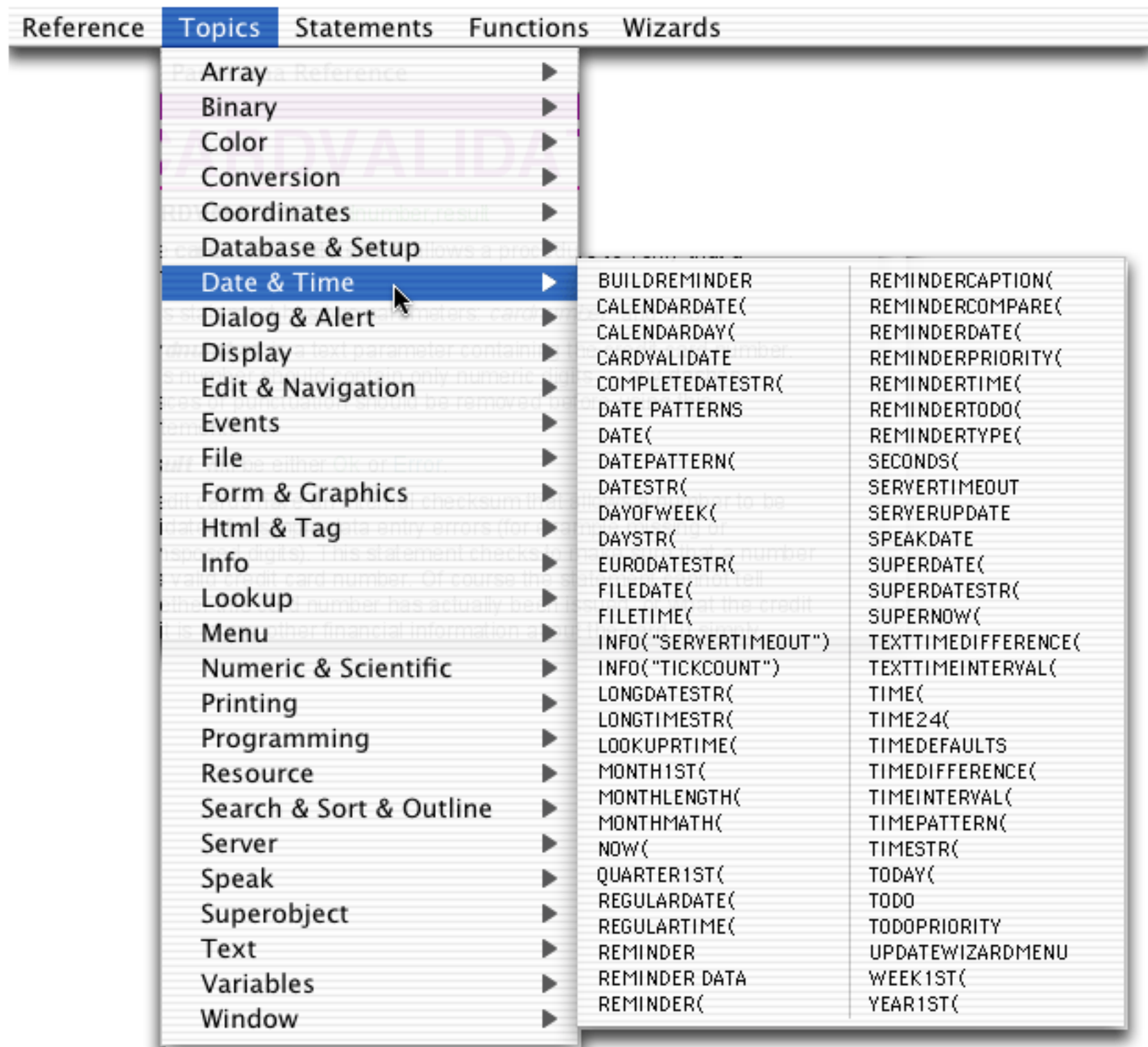


Repeating the search with the **Full Text Search** option turned on yields 14 matches. You can click on the match you are interested in.



Navigation Using the Topic, Statement and Function Menus

To jump directly to any topic use the **Topic**, **Statement** or **Function** menus. The **Topic** menu divides topics into about two dozen submenus. (Some topics may be display under more than one submenu, and some topics may not be listed under any topics.)



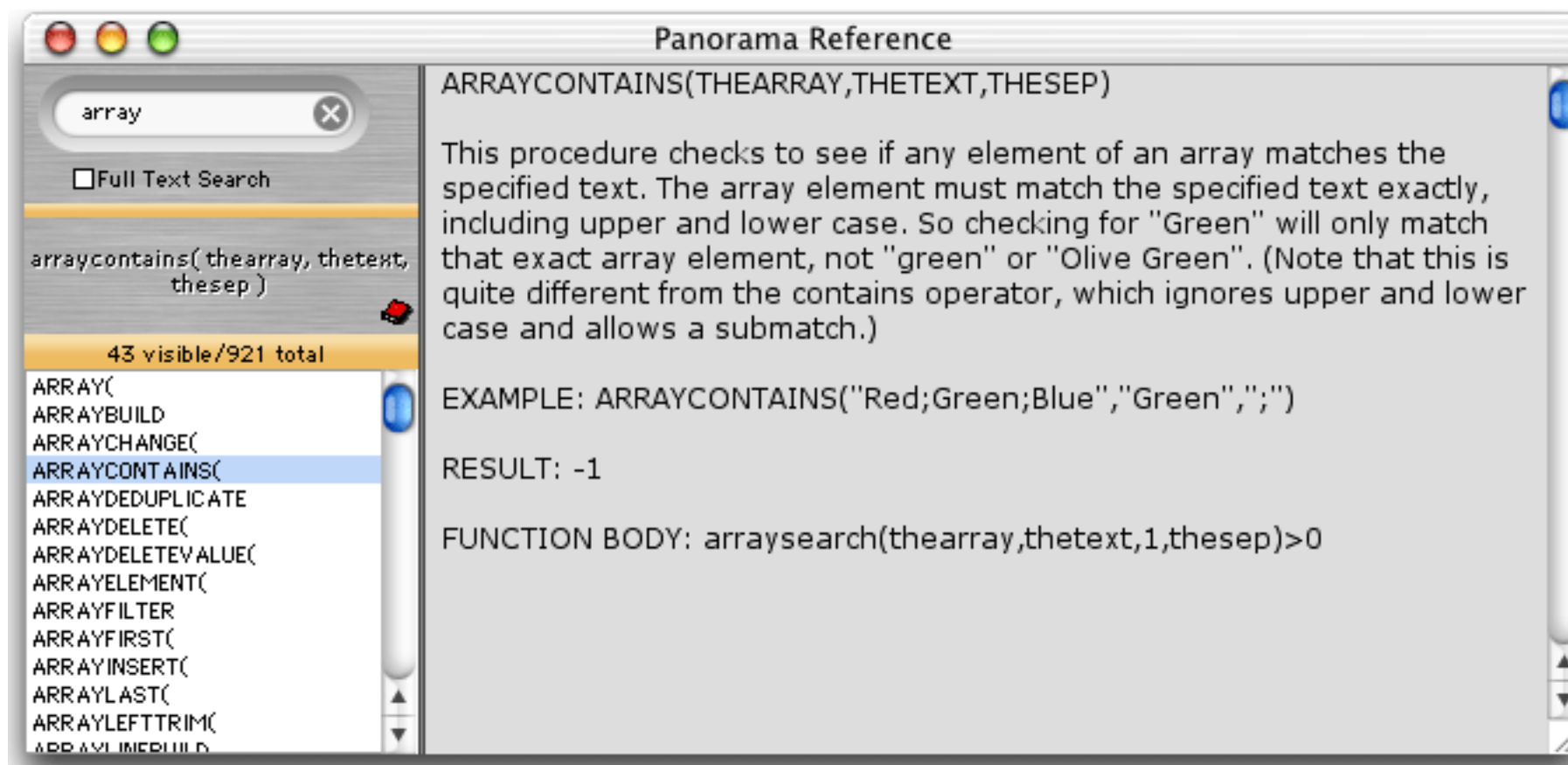
The **Statements** menu lists every statement in alphabetical order. The **Functions** menu lists every function in alphabetical order. Simply select a statement or function from one of these menus to jump to see the description of that topic in the topic panel.

Navigation Using HyperLinks

Like a web browser, the **Programming Reference** contains links from one page to other related topics. These links are underlined in the text. To jump any linked topic simply click on the underlined text.

Built In vs. Custom Statements and Functions

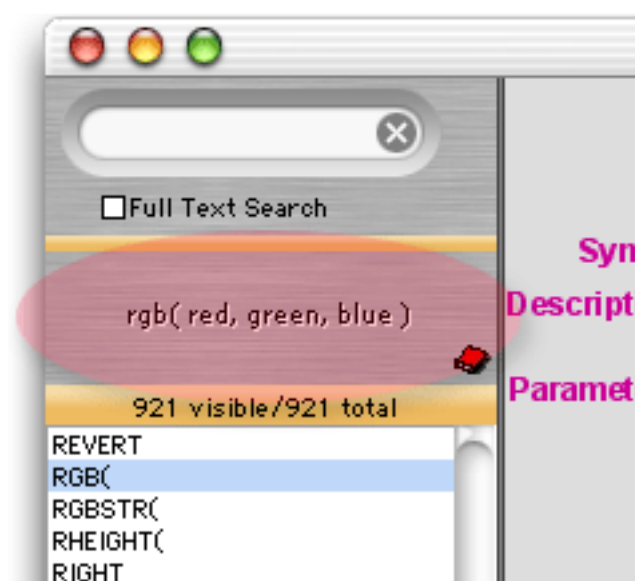
Panorama supports both built in and custom statements and functions. Several hundred custom statements and functions are included with Panorama, and these are also included as topics in the **Programming Reference** wizard. (You can also create your own custom statements and functions, but these are not included in the **Programming Reference** wizard.) For most custom statements and functions the topic panel uses a basic "plain text" format instead of the more graphical format used for built-in statements and functions.



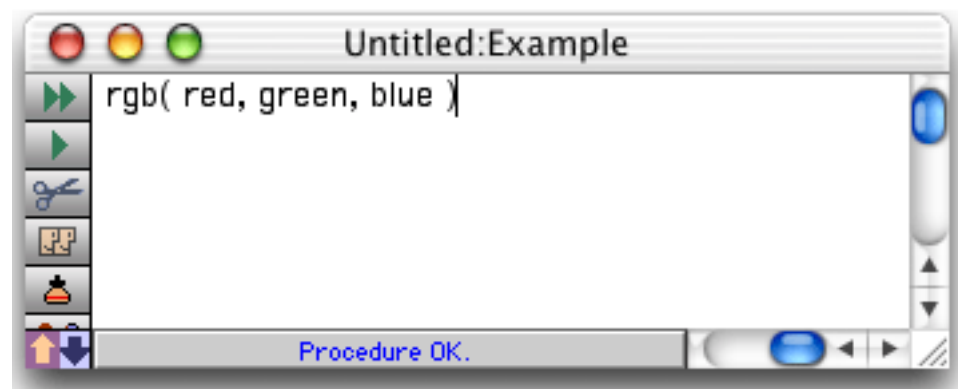
Don't adjust your set -- this plain text view is normal for custom statements and functions.

Using the Template Panel

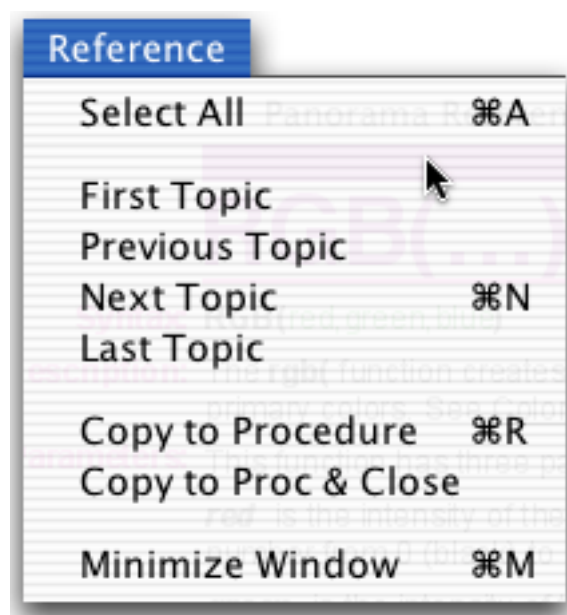
The template panel displays a sample that illustrates how this statement or function would be used in a formula or procedure. In this case the panel shows an example of the `rgb()` function, which has three parameters.



To copy the template into the topmost procedure window, hold down the **Control** key and click on the template panel. (If you are using a PC system you should right-click on the template panel.) The template will be pasted into the procedure at the current insertion point, and the procedure window will be brought to the front so that you can edit it further (for example, filling in the actual parameters).




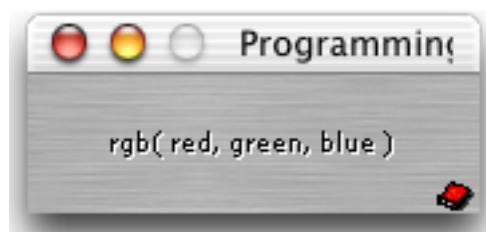
You can also copy the template into the procedure window using the Reference menu.




The **Copy to Procedure** command copies the template into the procedure and brings the procedure window forward (exactly like control-clicking on the template panel). The **Copy to Proc & Close** command does the same, but also closes the **Programming Reference** wizard.

Minimizing the Programming Reference Wizard

In addition to its normal "wide-screen" view, the wizard can also be used in a minimized view. To minimize the wizard, either choose **Minimize Window** from the Reference menu or click on the  button in the template panel. As shown here the minimized wizard hides the search panel, topic list, and topic display panel.



Although you can't search for topics when the window is minimized, you can still use the **Reference**, **Topics**, **Statements** and **Function** menus. When you want to maximize the window again choose either the **Maximize Window** from the Reference menu or click on the  button.

Data Flow

The purpose of almost any program is to organize and channel data. Since Panorama is a database, this is even more true for programs written in Panorama. This section discusses the techniques for storing and manipulating data within a procedure.

Assignment Statements

An assignment statement computes a value (text or numeric) and stores that value somewhere. Unlike every other statement, an assignment statement has no specific keyword that identifies the statement. Assignment statements always have the format shown below:

```
<data storage location> = <formula>
```

The first part of the assignment statement is the **data storage location**. This is the final destination for the data that is being moved. In fact, sometimes the data storage location is simply called the **destination** of the assignment. The data storage location may be a variable, a field in the currently active record, or the clipboard.

The next part of the assignment statement is the equals symbol. This identifies this statement as an assignment statement.

After the equals symbol is the formula. The formula produces the data that will be stored in the data storage location. The formula may simply take a variable or field and pass it along, or it may process, calculate or filter the data before it passes it along to be stored in the data storage location.

Here's a simple assignment statement that takes the contents of B and moves it into A. After this statement is finished both A and B will contain the same value.

```
A=B
```

More complicated assignment statements may combine multiple fields or variables, and they may process the data in some way. An assignment statement may also take a constant value and store it. Here are some examples:

```
A=B*C
```

```
Name=upper(myName)
```

```
City="San Francisco"
```

In each case, the process is the same. First Panorama calculates the formula to produce a data value. Then it stores the data value in a data storage location.

Triggering Automatic Calculations

A database can be set up so that when a field is modified by the user, one or more formulas are automatically calculated (see "[Automatic Calculations](#)" on page 303 of the *Panorama Handbook*). When an assignment statement modifies a field, however, these formulas are not automatically calculated. This is to give the procedure programmer the ultimate control over all calculations that occur during the procedure.

If you as the programmer would like the automatic calculations to be performed during an assignment, add an extra equal symbol to the assignment. The two equal symbols must be adjacent with no spaces between them, like this:

```
PriceΩ==19.95
```

In this example, storing the value 19.95 will most likely trigger several additional calculations to compute the total for this line item and the total for the entire invoice.

The Define Statement

The **define** statement is a special kind of assignment statement. This statement defines a value for a variable, but only if that variable doesn't already have a value. In other words, this statement will initialize the variable if the variable's value has not been defined yet, but if the variable already has a value it will not touch the value.

The define statement has two parameters: the name of the variable and the value for the variable.

```
define <variable>,<value>
```

The example shown below will initialize the variables **DefaultAreaCode** and **TaxRate** unless they have already been initialized.

```
global DefaultAreaCode,TaxRate
define DefaultAreaCode,"714"
define TaxRate,4.25
```

The Set Statement

The **set** statement performs an assignment, much like an equals sign. However, the destination of the assignment can be calculated on the fly.

```
set destination,formula
```

This statement has two parameters: **destination** and **formula**. **Destination** is a formula that calculates the name of the field or variable that you want to modify. **Formula** calculates the value that will be placed into the destination.

Panorama normally copies data into fields or variables with an assignment. For example, this assignment statement copies the value **Westside** into the field (or variable) named **City**.

Assignment statements like **City="Westside"** work fine as long as it is known where the data needs to be copied into when the program is written. But sometimes this is not known, or it needs to change on the fly. The **set** statement essentially lets the left hand side of the **=** change on the fly.

The procedure below assumes that the current database contains a field for each day of the week: **Sunday**, **Monday**, **Tuesday**, etc. The example copies the variable **DepartureTime** into the field for the current day (the second line of the procedure calculates the name of the day).

```
set datepattern( today(),"DayOfWeek"),DepartureTime
```

Here is the same procedure rewritten without the **set** statement. This illustrates the power of the **set** statement, which in this case is doing the same work as the 17 statements below.

```
local dayName
dayName= datepattern( today(),"DayOfWeek" )
case dayName="Sunday"
  Sunday=DepartureTime
case dayName="Monday"
  Monday=DepartureTime
case dayName="Tuesday"
  Tuesday=DepartureTime
case dayName="Wednesday"
  Wednesday=DepartureTime
case dayName="Thursday"
  Thursday=DepartureTime
case dayName="Friday"
  Friday=DepartureTime
case dayName="Saturday"
  Saturday=DepartureTime
endcase
```

The FormulaValue Statement

Like the `set` statement, the `formulavalue` statement calculates the result of a formula and puts it somewhere. The `formulavalue` statement gives you more control over how errors are handled, allows you to specify the formula using a variable, and allows you to specify what database is to be used for the calculation (an assignment or set statement always uses the current database).

```
formulavalue destination,database,formula
```

This statement has three required parameters: `destination`, `database` and `formula`. The `destination` is the name of the field or variable that will receive the result. If a field is specified it must be in the current database, even if you have specified that the formula be calculated using a different database.

The `database` parameter is the database to be used for performing the calculation. Any field values specified in the formula will be obtained from the current record in the specified database. If no database is specified ("") the current database will be used.

The `formula` parameter is the formula to be calculated. Unlike an assignment statement, the formula is contained in a formula, allowing you to change the formula on the fly. If the formula doesn't need to be changed you must enclose it in quotes, otherwise you can store the formula in a variable. Unlike most statements the `formulavalue` statement will not stop if an error occurs in evaluating the formula. To find out if an error occurred you must check the `info("error")` function.

This simple example performs a calculation using the `Checkbook` database, which may or may not be the current database (it does have to be open). There will be no window flashing or other visual artifacts even if another database is currently on top.

```
local myFormula,lineBalance
myFormula="Debit-Credit"
formulavalue lineBalance,"Checkbook",myFormula
```

The `formulavalue` statement is especially useful in situations where the user can enter their own formula. This example prompts the user to enter a formula. The result of the formula is calculated and displayed, unless there is an error, in which case that is displayed.

```
local myFormula,myAnswer,myError
gettext "Enter Formula:",myFormula
formulavalue myAnswer,"",myFormula
myError= info("error")
if myError=""
    message "The answer is: "+ constantvalue( myAnswer)
else
    message info("error")
endif
```

The `formulavalue` statement is also useful for validating user supplied formulas. If the `formulavalue` statement can process the formula without an error, then so can other statements like `select` or `arraybuild`. In this case we don't really care what the resulting value from the formula is, as long as it is a number (integer) and there is no error.

```
local myFormula,myAnswer,myError
gettext "Enter Formula:",myFormula
formulavalue myAnswer,"",myFormula
myError= info("error")
if myError <> ""
    message info("error")
    rtn
endif
if datatype( "myAnswer") <> "Integer"
    message "Formula must calculate a true/false answer"
    rtn
endif
execute " select "+myFormula
```

Variables

A variable is a place in the computer where an item of data can be stored, kind of like a storage bin for a value. Variables may be created by procedures or by SuperObjects. Most procedures will use one or more variables to hold and transfer data as the program runs. Use a variable whenever you need to store a single data item so that you can use it later. Unlike a field, the value variable doesn't change as you move from record to record, or, in the case of a global variable, even when you move from database to database.

Creating a Variable

Panorama has five different statements for creating variables within a procedure. The most common one is the `local` statement (see “[LOCAL](#)” on page 5489 of the *Panorama Reference*), which generates temporary variables that only last until the procedure is finished. This statement should be followed by a list of the variables to be created, with each name separated from the next by a comma. This example creates four variables. Because these variables were created with the `local` statement they are called **local variables**.

```
local alpha,gamma,delta,sigma
```

Creating a variable is kind of like surveying a lot on empty land. Once the land is surveyed you know where it is, but the land is still empty until something is built on it. A new variable is like an empty plot of land that has just been surveyed — it has an address (the name) but it doesn't have any data yet.

By the way, Panorama allows any sequence of characters to be used as a variable name. However, if the variable name contains any punctuation (including spaces) it must be surrounded by the chevron characters « and ». (On the Macintosh press **Option-** to create the « chevron character and **Shift-Option-** to create the » chevron character. On Windows systems press **Alt-0171** to create the « chevron character and **Alt-0187** to create the » chevron character.) Here are some examples of typical variable names:

```
x
birthDay
Counter
«Tax Rate»
«PrimeRate%»
```

A variable name must be spelled exactly the same way every time, including upper and lower case. The variable name `birthDay` is not the same as `Birthday` or `birthday`. In fact, you could create three different variables using these three different names (although this is not recommended because it would be very confusing).

By the way, it's always ok to use chevrons around a variable name, even if the name doesn't have any punctuation. «`Counter`» is exactly the same as `Counter`, and they can be used interchangeably. So if you have any doubts about whether or not chevrons are necessary, go ahead and use them. No harm, no foul.

Note: Some programming languages require you to create all variables first, before any other statements. Panorama isn't that picky. You can create new variables anywhere in the program. Here is a procedure that creates four variables, does some work, then creates two more variables.

```
local alpha,gamma,delta,sigma
  alpha=4
  gamma="blue"
  delta=alpha*3
  sigma=delta/alpha
local epsilon,omega
  epsilon=0.01
  omega="z"
```

By the way, the indentation is not necessary, it's simply to make the local statements easier to see.

Assigning a Value to a Variable

Once a variable has been created you can assign a value to it. This is done by putting the variable on the left side of an assignment statement (see “[Assignment Statements](#)” on page 243) like this.

```
alpha=4  
  
gamma="blue"  
  
delta=alpha*3  
  
sigma=delta/alpha
```

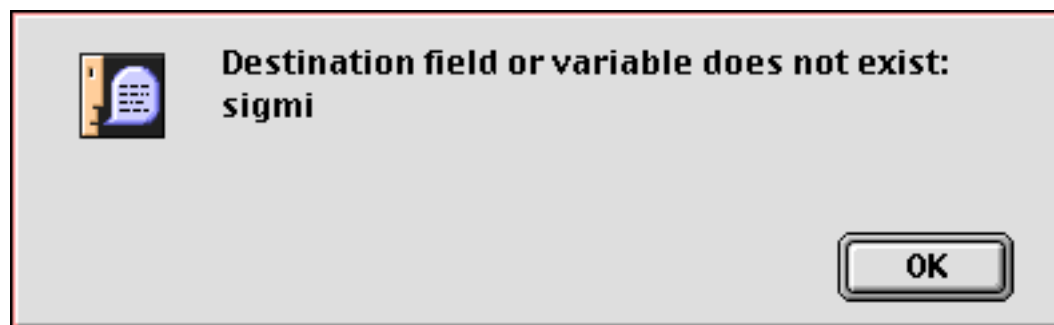
A new value can be assigned to a variable at any time. You can assign a value to a variable once or over and over again a million times. The new value does not have to have anything to do with the old value — you can store a number in a variable that originally held text or vice versa — Panorama doesn’t care. This line of text assigns the number **400** to the **gamma** variable, which originally had a text value stored in it.

```
gamma=400
```

Some programming languages allow you to assign a value to a variable without having to create the variable first. Panorama’s programming language does not allow this. For example, this procedure creates four variables, then attempts to store a value into a fifth variable, **sigmi**.

```
local alpha,gamma,delta,sigma  
sigmi=200
```

When you run this procedure Panorama will complain. Picky picky picky!



In this case the problem is probably a typo, and the variable name in the assignment needs to be corrected. If this really is a separate variable you must create it first.

Using a Variable in a Formula

Once a variable has been assigned a value you can use it in a formula. A variable may be used anywhere a field or constant may be used. Here are some typical examples.

```
alpha*sigma  
  
gamma+" action"  
  
4*alpha/(delta+20)
```

See “[Formula Grammar](#)” on page 42 to learn more about using variables in a formula.

The Birth and Death of a Local Variable

When a lot is surveyed, that lot usually exists more or less forever (barring wars or natural disasters). A local variable, however, is not nearly that permanent. In fact, local variables only exist until the end of the procedure that they are created in. When the procedure finishes, Panorama checks to see if it created any local variables. If it did, the values in these variables are dumped out and all record of the variables are destroyed, just as if they had never existed in the first place. It's kind of as if you went down to the county recorder's office and burned all the survey records for a tract of land.

Why are the local variables destroyed? Two reasons. First, they take up memory that is no longer needed and can be used for other things. Secondly, this allows different procedures to use the same variable names without having to worry about conflicting with each other. Suppose procedure A and procedure B both have a local variable named **gamma**. Since the variable is destroyed when each procedure finishes, neither procedure needs to worry about the other. Each happily creates and uses its own copy of **gamma**, which is then destroyed before it can interfere with any other procedure.

Note; If you are an experienced programmer you may be wondering about recursion at this point. If you can't resist, you can jump ahead to "[Recursive Subroutines](#)" on page 273.

Long Life Variables

Sometimes, of course, you won't want a variable to be destroyed when the procedure is finished. Panorama actually has five different kinds of variables, each with different life cycles (local, window, fileglobal, global and permanent). You've already learned about local variables. The next most common type of variable is a **fileglobal variable**, which I'm sure you'll be surprised to learn is created with the **fileglobal** statement (see "[FILEGLOBAL](#)" on page 5227 of the *Panorama Reference*). Just as with the local statement, the fileglobal statement is followed by a list of variables to create. This statement creates two variables.

```
fileglobal Speed,Direction
```

Unlike a local variable, a fileglobal variable isn't destroyed when the procedure is finished. It hangs around and can be used over and over again. However, fileglobal variables don't last forever. When the database is closed, the values in these variables are dumped and the variables themselves are destroyed.

As you might guess, a **permanent variable** has a very long life. However, the life of a permanent variable is not continuous but interrupted. When the database is closed any permanent variables associated with that database are destroyed. However, before the variables are destroyed the values are stored in the database itself. When the database is re-opened later Panorama automatically re-creates the permanent variables again. (Note: You must save the database. Just as with data in database fields, Panorama only saves permanent variables when the database is saved.)

Another type of long life variable is a **global variable**. A global variable is not destroyed even when the database that created it is closed. It's almost immortal. However, when you **Quit** from Panorama, that's the end of the road for global variables. They are not re-created automatically the next time Panorama opens.

A specialized kind of variable is a **window variable**. This kind of variable is attached to whatever window was open and on top when it was created. When that window is closed, the variable is dumped. Poof! Window variables are usually used with clonable forms (see "[Window Clones](#)" on page 457).

Destroying a Variable

If necessary you can use the **undefine** statement to destroy any variable at any time (see "[UNDEFINE](#)" on page 5866 of the *Panorama Reference*). When you use this statement the variable (or variables) is completely destroyed as if it had never been created in the first place. This example destroys the variables **Speed** and **Direction**.

```
undefine Speed,Direction
```

You can destroy any kind of variable. If a variable can be accessed, it can be destroyed. Before you destroy a permanent variable, however, you should first make it un-permanent, like this:

```
unpermanent timeStamp
undefine timeStamp
```

It's rarely necessary to destroy a variable yourself. Panorama automatically destroys local variables when the procedure opens, destroys fileglobal variables when the file is closed, and destroys windowglobal variables when the window is closed.

Variable Accessibility

Just because a variable exists doesn't mean you can access it. Many types of variables are "attached" to a file or a window and are only available when that file or window is active. When the file or window isn't active, these variables are "dormant." They still exist (and take up memory), but you cannot access or modify them until the file or window they are attached to becomes active again.

Fileglobal and **permanent** variables are attached to the file that was active when they were created. When this file is open and on top, the variables are accessible and can be used in a formula or modified with an assignment statement. When some other file is on top these variables are dormant and cannot be used.

The beauty of this system is that it allows you to create variables without worrying about conflicting with other databases. Consider the two fileglobal variables created in the previous section, **Speed** and **Direction**. What if some other open database also has variables with these names? As long as both databases use fileglobal variables instead of global variables (see below) they'll both be all right. Each will have their own separate **Speed** and **Direction** variables. When database A is active its variables will also be active while B's are dormant. When database B is active A's variables become dormant. Essentially Panorama will keep two completely separate sets of **Speed** and **Direction** variables, each with their own values.

Windowglobal variables are attached to the window that was active when they were created. You can have multiple cloned windows that use the same variable name but actually each has its own separate variable that is only active when the window is on top (see "[Window Clones](#)" on page 457).

Global variables are always active, no matter what file or window is on top. This is great if you need to have data that is accessible anywhere at any time. But be careful! If two different databases use the same global variable they had better be co-operating with each other! Remember that you may open databases that were created by other people. Who knows what global variable names they will use? If you do need to use global variables we recommend that you use very long descriptive names like **ProTechSpeed** or **AcmeSalesTaxRate**. Names like this are more likely to be unique and not conflict with anyone else's global variable names.

Accessing "Dormant" Variables

Fileglobal, permanent and window variables are normally "dormant" when the file or window they are associated with is not open and on top (see "[Variable Accessibility](#)" on page 250). However, it is possible to access the values in these dormant variables with special functions. The **grabfilevariable()** function (see "[GRABFILEVARIABLE\(\)](#)" on page 5328 of the *Panorama Reference*) can grab the value of a value even if it is dormant. The function below will grab the value of the **Speed** fileglobal variable even if the **Transpac Race** database is not on top (it must be open, however). Notice that the variable name ("**Speed**") must be in quotes.

```
grabfilevariable("Transpac Race","Speed")
```

The **grabwindowvariable()** function is similar (see "[GRABWINDOWVARIABLE\(\)](#)" on page 5329 of the *Panorama Reference*) except that it grabs the value of windowglobal variables for windows that are not on top.

“Hidden” Variables and Fields

You may wonder what happens if two or more variables are both accessible and have the same name. For example, what if the current database has a fileglobal variable named **Grok** and there is also a global variable named **Grok**. In this case the global variable is “hidden” behind the fileglobal variable and cannot be accessed. The global variable will remain hidden as long as this database is active. The table below shows how different types of variables can hide other types of variables.

Type of Variable	Hides these types (if the name is the same)
local	all other types of variables and fields
windowglobal	fileglobal variables, permanent variables, global variables and fields
fileglobal or permanent	global variables and fields
global	database fields
field	nada!

As the table shows, variables can also hide database fields if they have the same name. This can especially be a problem with global variables, which can interfere with any field in any database. (By interfere we mean that the field will not be accessible in a formula.) Be sure to avoid global variable names like **Name**, **Address**, **State**, **Amount**, or any name that might be likely to be used as a database field.

Accessing Variables In Form Objects (Text or Images)

Several different form objects can display the results of a formula (all references are to the *Panorama Handbook*).

Graphic Objects That Can Display Variables Using a Formula
Auto-Wrap Text (see “ Displaying Formulas in Auto-Wrap Text ” on page 602)
Text Display SuperObjects (see “ Text Display SuperObjects™ ” on page 608)
Text Editor SuperObjects (see “ Text Editor SuperObject ” on page 639)
Word Processor SuperObject (see “ Merging Data into Word Processing Documents ” on page 707)
Flash Art and Super Flash Art (see “ Flash Art™ ” on page 750)
Data Button (see “ Data Buttons ” on page 837)
Pop-Up Menu Buttons (see “ Pop-Up Menus ” on page 860)
List SuperObjects (see “ List SuperObjects ” on page 879)

As long as it is accessible (see “[Variable Accessibility](#)” on page 250) a variable can be used in the formulas for any of these objects. For example, a Text Display SuperObject may display any global variable, any fileglobal or permanent variable created in the same file, or any windowglobal variable created in the same window.

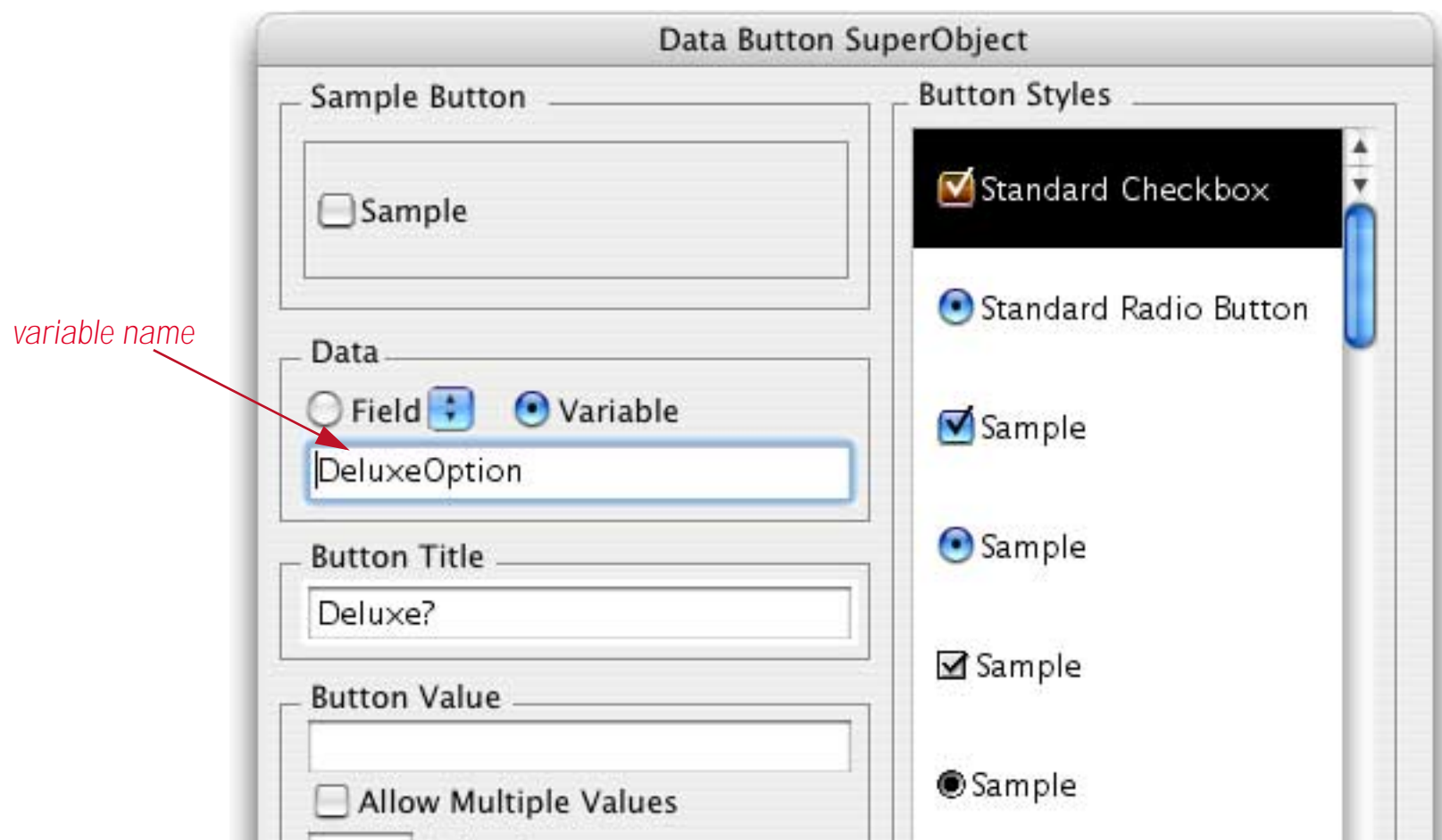
Since a local variable is created and destroyed within each procedure local variables are not accessible to formulas in form objects. If you want to display a variable in a form it must be a windowglobal, fileglobal, permanent or global variable.

Creating Variables with a SuperObject

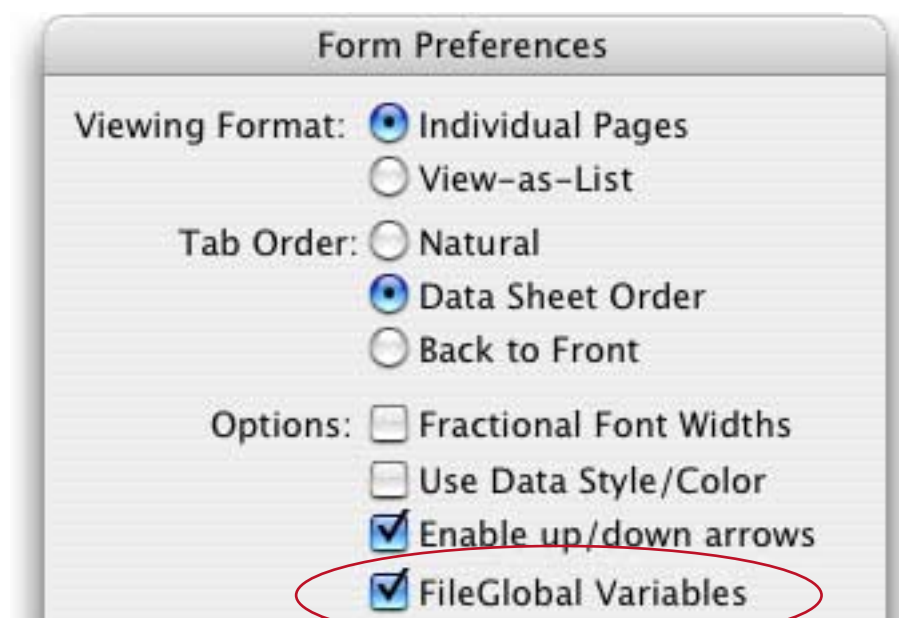
Variables are usually created in a procedure (see “[Creating a Variable](#)” on page 247). However, several different types of SuperObjects have the option of linking to a variable or a field, and these objects will automatically create the variable if it does not exist. When a variable is created by a SuperObject it is always a global or fileglobal variable and is initialized to empty text. SuperObjects that can create variables include the Text Editor (see “[Text Editor SuperObject](#)” on page 639 of the *Panorama Handbook*), Word Processor (see “[Word Processor SuperObject](#)” on page 673 of the *Panorama Handbook*), Data Button (see “[Data Button SuperObjects™](#)” on

page 838), Pop-up Menu (see “[Pop-Up Menu SuperObjects™](#)” on page 860 of the *Panorama Handbook*), List (see “[List SuperObjects](#)” on page 879 of the *Panorama Handbook*), Sticky Button (see “[Sticky Push Button SuperObjects™](#)” on page 855 of the *Panorama Handbook*) and Scroll Bar (see “[Scroll Bars](#)” on page 979 of the *Panorama Handbook*).

For example, suppose you are working on a form and create a checkbox using the **Data Button** tool (see “[Data Buttons](#)” on page 837 of the *Panorama Handbook*). You select the **Variable** option and type in the variable name **DeluxeOption**, as shown in this illustration.



When the **OK** button is pressed, Panorama checks to see if you have already created a variable named **DeluxeOption**. This may be a global variable, a fileglobal or permanent variable (in this database) or a windowglobal variable (in this window). If the variable has already been created, Panorama will simply use it. But if there is no such variable, Panorama will create it as a global or fileglobal variable. The default is a global variable unless the **FileGlobal Variables** option is set in the **Form Preferences** dialog (Setup menu). Except for how it was created, this variable is just like any other variable and can be used freely in procedures and formulas.



Panorama actually creates the variable the first time it displays this object. If you shut down Panorama and then later re-open it, the variable will be created the first time the form is displayed. If database has been saved with the **Save Window Position** option turned on (see “[Saving Window Positions](#)” on page 64 of the *Panorama Handbook*) so that this form opens automatically the variable will be created immediately when the file is first opened.

Permanent Variable Tips

When the **permanent** statement creates a permanent variable, it really creates two variables: one in memory and one in the current database. The one in memory is an ordinary fileglobal variable. Whenever the database is saved, Panorama copies the contents of the fileglobal variable into the copy of the variable in the database itself, then saves the database. Just like any other data, the contents of the permanent variable are not saved unless the database itself is saved. However, if you have not made any other changes to the database, Panorama will not warn you if you attempt to close a database without saving changes to the permanent variable.

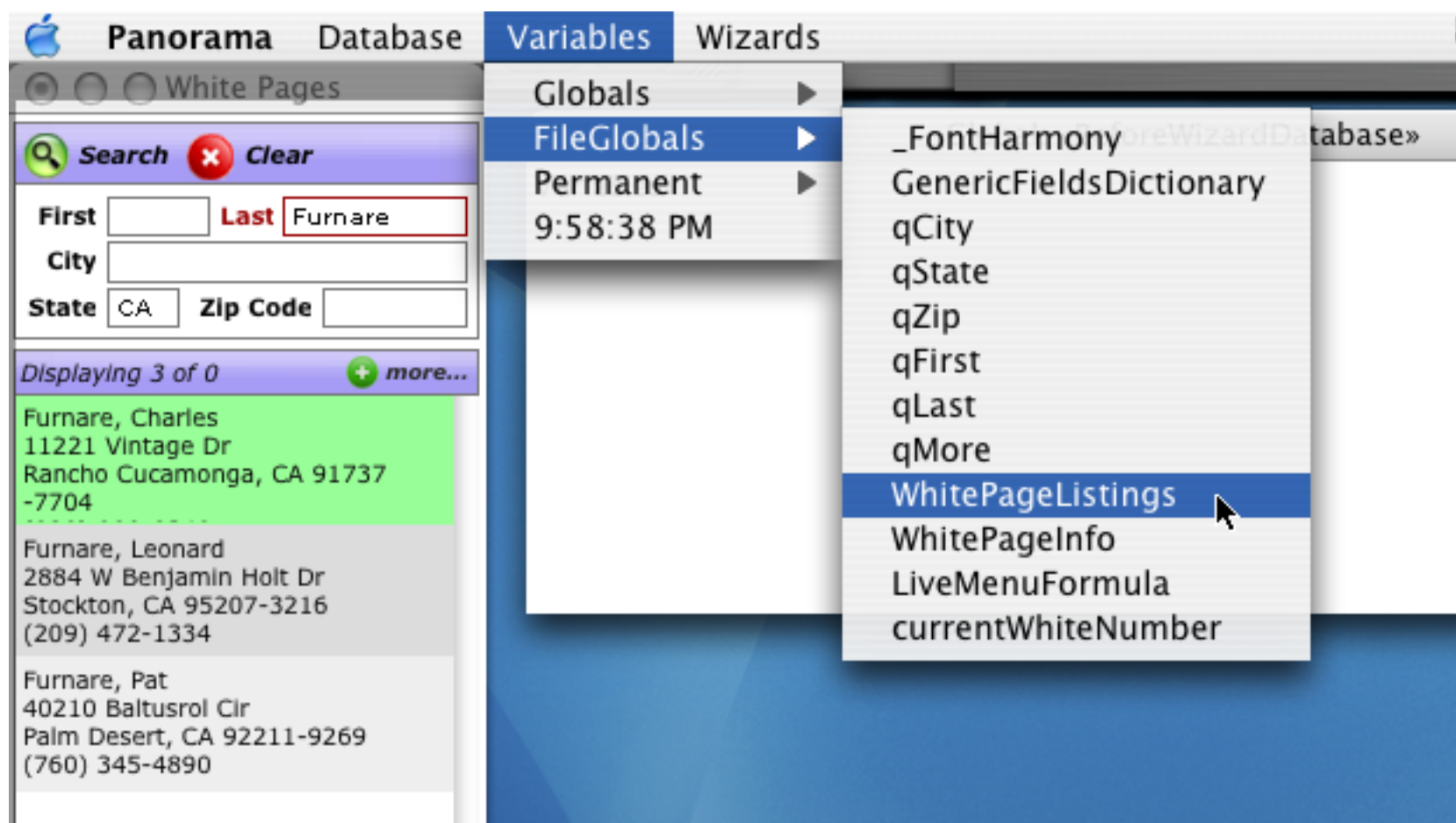
Whenever a database is opened, Panorama automatically creates fileglobal variables for any permanent variables associated with that database. Next it copies the values from the database into the fileglobal variables. The variables are now ready to use.

If you ever want to make a permanent variable un-permanent, use the **unpermanent** statement, which is followed by a list of variables you want to make unpermanent. This statement doesn't make the variables go away, but they will no longer be permanent. The **unpermanent** statement only affects variables that are permanent in the current database. The example below changes two permanent variables back into regular (non-permanent) global variables.

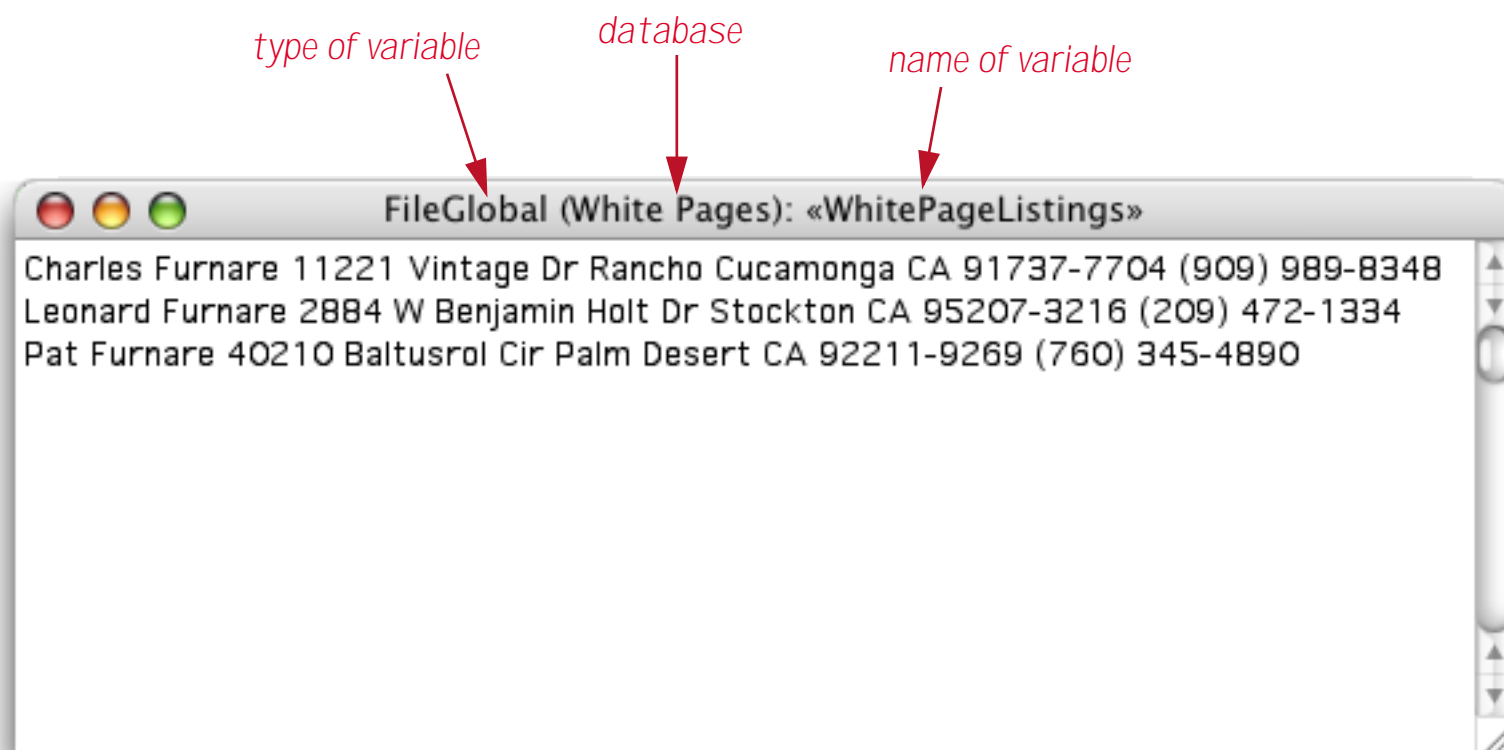
```
unpermanent myAreaCode,myZipCode
```

Displaying and Changing Variables

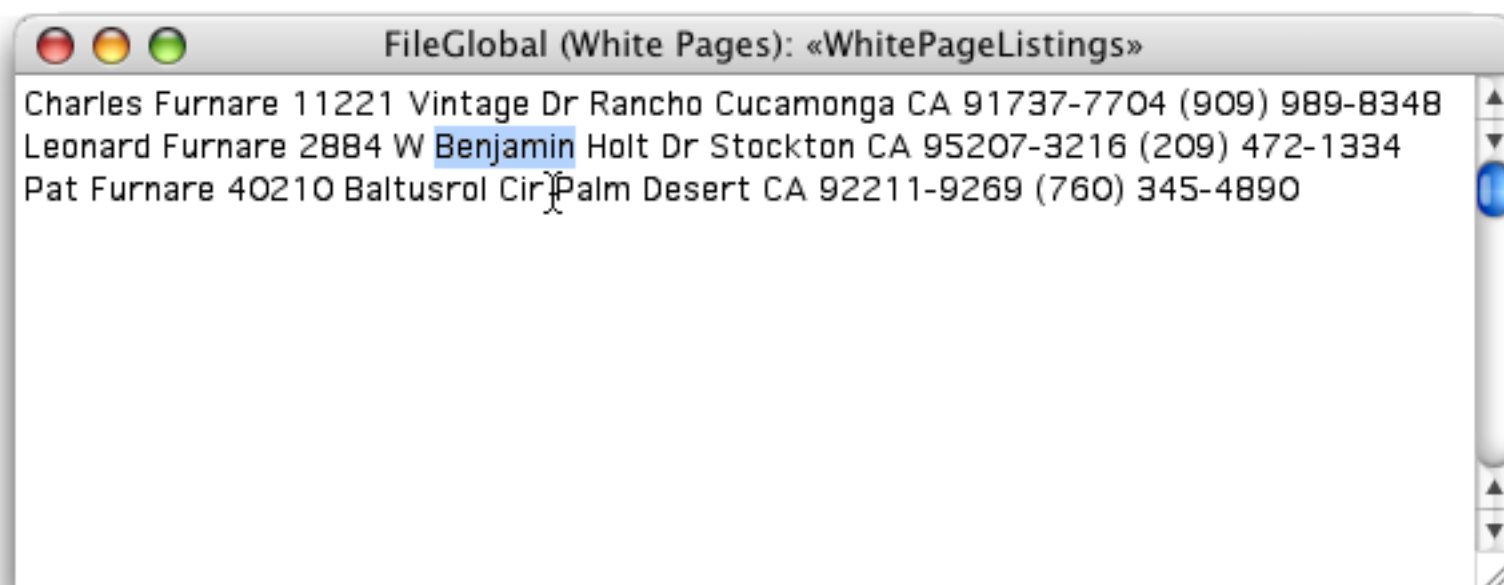
Global, **fileglobal** and **permanent** variables can be displayed and modified with the **Variables** wizard (in the Developer Tools submenu of the Wizard menu). To display a variable open the wizard and then pick from one of the three variable submenus.



The wizard window will show the value of the variable. (The window title will show the type of variable, the database it is associated with and the actual name of the variable.



If you are absolutely sure you know what you are doing, you can click on the variable to edit the value.



When you press the **Enter** key the variable will be changed. Press **Command-Period** if you decide not to change the value (**Control-Period** on PC systems).

If you want to view variables in another database use the **Database** menu to select the database that contains the variables you want to examine.

Control Flow

As it runs a procedure, Panorama usually starts with the first statement in a procedure and works its way down. However, Panorama is not limited to this kind of linear approach. The procedure can be designed to make comparisons or decisions and take different steps depending on the result. For example, the procedure might decide to skip one or more steps, or it might decide to repeat a sequence of steps more than once. The procedure may even decide to trigger another procedure to help it complete its job. Programmers call this decision making process control flow, because it controls and possibly alters the flow of statements dynamically as the program is running.

The ability to change the flow of steps “on-the-fly” is the key to programming. A simple example may help make the utility and power of this concept more clear. Suppose you want to create a procedure that adds fifty new records to the end of the database. You could simply create a procedure with the `AddRecord` statement repeated fifty times. Of course this is inconvenient, and what if you wanted to create 2500 new records? Instead of repeating the `AddRecord` statement over and over, you can write a program that repeats a single `AddRecord` statement over and over until the proper number of records have been added.

```
loop
  addrecord
until 50
```

The program is much shorter than if you had literally repeated the `AddRecord` statement 50 times, and can be changed easily if you want to add a different number of new records.

Now suppose you want to change this program so that as it adds new records, it alternately puts the word `Black` or `Gold` into the `Color` field of each record. This requires the program to make a decision for each record—is this record even or odd? There are several ways this could be programmed in Panorama, The example below shows just one of them. In this example you’ll notice that all seven steps between loop and until are being repeated 50 times.

```
local nextColor
nextColor="Black"
loop
  addrecord
  Color=nextColor
  if nextColor="Black"
    nextColor="Gold"
  else
    nextColor="Black"
  endif
until 50
```

Although this program is quite simple, it illustrates the basic elements of control flow.

True/False Formulas

In Panorama as in most programming languages, control flow decisions are made on the basis of formulas that are either true or false. The most basic true/false formula compares two values to see if they are equal.

```
PaymentMethod="C.O.D."
```

This formula will compare the value in the field `PaymentMethod` with `C.O.D.` The result will be true if `PaymentMethod` is `C.O.D.`, and false if it contains anything else (for example `Check`, `Cash`, `Visa`, etc.). To learn more about true/false formulas and how to create them see “[True/False Formulas](#)” on page 124.

Equals Comparison vs. Assignment

If you have been paying attention you undoubtedly noticed that the formula in the previous section looks exactly like an assignment. Why doesn't this formula

```
PaymentMethod="C.O.D."
```

assign the value **C.O.D.** to the field **PaymentMethod**? At first glance this may appear ambiguous...the same formula is used to compare two values and to assign a value. How do we know when we are assigning and when we are comparing? The answer lies in the context in which the formula is found.

In a procedure, an assignment is always by itself, not part of a larger statement. A true-false formula is always part of another statement, for example **if**, **case**, **until**, **while**, **stoploopif**, **repeatloopif**, **find**, **select**. Here's an example that shows two formulas that look almost the same, but one is a true-false formula and one is an assignment.

```
if PaymentMethod="C.O.D."
  ShippingMethod="UPS"
endif
```

The first formula, **PaymentMethod="C.O.D."**, is part of the **if** statement. Because it is part of the **if** statement this formula means: Is the field (or variable) **PaymentMethod** equal to **C.O.D.** (true/false)?

The second formula, **ShippingMethod="UPS"**, is not part of any statement, but stands alone, so this is an assignment. The statement means: Take the value **UPS** and copy it into the field or variable named **ShippingMethod**.

If an assignment has more than one equals sign, the first equals sign is for the assignment and the rest are for comparisons. The example assignment below compares B and C. If they are equal (true) the value -1 will be copied into A. If they are not equal (false) the value 0 will be copied into A.

```
A=B=C
```

In other words, **A** becomes the result of the comparison between **B=C**.

True/False Values

For purposes of calculation, Panorama treats true and false as numbers: true is -1 and false is zero. Like any other number, you can store a true/false value in a field or variable and then use it later. The example below calculates whether a person is a teenager, then uses that information later.

```
local Teenager
Teenager=Age≥13 and Age<20
...
if Teenager
  Price=4.50
else
  Price=6.00
endif
```

Notice that the **if** statement doesn't need to compare, it simply uses the result of the comparison that was calculated earlier. In fact, the **if** statement (and all other statements that use true/false logic) can use any formula that produces a numeric integer result. The value 0 will be regarded as false, and any non-zero value will be regarded as true. The example below will be true if the length of the name is non-zero.

```
if length(Name)
  yesno "Is this a home address?"
  ...
endif
```

The first line of this example could also have been written **if length(Name)<>0**. The result is the same either way.

IF Statements

The basic building block for making decisions in a Panorama program is the **if statement**. The **if** statement will skip over the next few statements (up to the next **endif** statement) if the true/false formula is false. Here's a simple example.

```
if City=""
  City="Pismo Beach"
  State="CA"
endif
message City+", "+State
```

Depending on what's in the **City** field, this procedure can work one of two ways. If the **City** field is empty, the true/false formula **City=""** will be true, so the procedure will perform the assignments **City="Pismo Beach"** and **State="CA"**. But if the **City** field is not empty, the true/false formula will be false, and Panorama will skip past the **endif** to the message statement.

In this example there are two statements between the **if** and **endif** statements. These two statements will be skipped if the formula is false. However, there is no limit to the number of statements that may be between the **if** and **endif**. Just make sure that there is always a matching **endif** for every **if**. Although it is not required, indenting the statements between the **if** and the **endif** usually makes the procedure easier to read and understand.

ELSE Statements

The **else statement** turns the **if** statement into a two way operation: if true, do this, otherwise, do that. You could do this with two **if** statements in a row, but the **else** is simpler.

To use the **else** statement, place it between the **if** and **endif** statements. If the true/false formula is true, Panorama will perform the statements from the **if** up to the **else** and skip the statements from the **else** up to the **endif**. If the true/false formula is false, Panorama will skip the statements from the **if** up to the **else** and perform the statements from the **else** up to the **endif**.

The example below calculates sales tax and shipping for both in-state and out-of-state purchases.

```
if State="CA"
  SalesTax=0.08
  Shipping=2.50
else
  SalesTax=0
  Shipping=5.00
endif
```

If the state is **California**, the sales tax is 8% and shipping is \$2.50. But if the purchase is from any other state, the sales tax is zero and shipping is \$5.00. In this example more or less the same statements are used in both halves of the **if/else**, but this is not necessary. The two sections could be completely different.

Nested if Statements

Panorama is not limited to one **if** at a time. Panorama can make a decision, execute some more statements, and then make a subdecision. Since the inner **if endif** pair is completely surrounded by the outer pair, this is called nesting.

```
local CardLength
if PaymentMethod="Credit Card"
  CardLength=length(CardNumber)
  if CardLength<13 or CardLength>16
    message "Sorry, invalid credit card number."
  endif
endif
```

If the `PaymentMethod` is not `Credit Card`, the procedure will skip all the following statements and do nothing. But if the `PaymentMethod` is `Credit Card`, the procedure will continue and calculate the `CardLength` variable. The second `if` statement checks the card length. The `message` statement will only be performed if both `if` statements are true.

Error Handling with `if error`

There are literally hundreds of different errors that can occur while a procedure is running. Of course you'll want to eliminate all of the errors in the procedure itself, but many errors are the result of circumstances beyond the programmers control. A file can fail to open because it was placed into the wrong folder, the user can enter the wrong data type into a formula, the list is endless. When such an error occurs, Panorama's normal response is to display an error message and stop the procedure immediately. (In addition, if the procedure window is open, Panorama will attempt to highlight the location of the error.)

If you want to create a database that operates professionally, simply stopping the procedure half finished if there is an error may not be acceptable. Instead, you may want your procedure itself to trap the error and try to correct it, if possible. At a minimum, you may be able to display an error message that is more relevant to an untrained operator than Panorama's general purpose error messages.

To trap errors, use the `if error` statement (two words - there **must** be a space). This statement must be placed immediately after the statement that you are worried might cause an error. For example, suppose you have a procedure that appends a file with the `openfile` statement. If the file is missing or has been moved an error will occur. This example checks for that error, and if the error occurs, asks the user to enter a new file name. The procedure will keep trying until the file is opened successfully or the user gives up and enters an empty name.

```
local txFileName
txFileName="New Transactions"
loop
  openfile "+"+txFileName
  if error
    gettext "Enter the file name",txFileName
    reloopif txFileName≠""
  endif
while 1≠1
if txFileName="" stop endif
/* further processing of the new transactions, below */
...
```

A very useful trick for `if error` is checking to see if a global variable has been initialized with a value (see "[Assigning a Value to a Variable](#)" on page 248). If the variable has already been initialized with a value, you don't want to change that value, but if it has not been initialized, you do want to set the value. The example below checks the `AreaCode` global variable to see if it has already been set by another procedure. If it has, the statement `xTest=AreaCode` will work perfectly. But if `AreaCode` doesn't have a value yet, this statement will produce an error. The `if error` statement traps the error and sets the `AreaCode` variable to `714`.

```
fileglobal AreaCode
local xTest
xTest=AreaCode
if error
  AreaCode="714"
endif
```

If you have a lot of variables it may not be necessary to test each one, as long as they are initialized as a group by any procedure that sets them up. If they are initialized as a group you can just test one variable, then if it has not been initialized you can initialize the entire group.

`if error` must be used by itself, you cannot combine other conditions. For example, the statement:

```
if error and info("modifiers") contains "shift"      /* WILL NOT WORK !! */
```

will NOT work. To get this effect you must nest a second if statement inside the if error, like this.

```
if error
  if info("modifiers") contains "shift"
    ...
  endif
endif
```

Another way to handle errors is with the **onerror** statement, which allows you to change Panorama's default behavior for handling an error. See "[Catching Program Errors \(Especially for Web and other Server Applications\)](#)" on page 288 for details on this statement.

CASE Statements

If a program needs to select one (and only one) option out of many, the **case statement** is the way to go. Like the **if** statement, the **case** statement uses a true-false formula to decide whether or not to perform the following statements. But unlike the **if** statement, which is used alone, the **case** statement is always used in groups. Panorama checks the true-false formula for each **case** statement. If it is false, it skips to the next **case** statement. If it is true, it performs the statements until the next **case** statement. Then it skips past all the rest of the **case** statement to the **endcase** statement.

After all the **case** statements, you may optionally add a **defaultcase** statement. This will pick up any left-overs that weren't included in any of the other cases.

The example below shows how the case statement can be used to divide people up into five age groups.

```
case Age<5
  AgeGroup="Pre-School"
case Age<13
  AgeGroup="Youth"
case Age<20
  AgeGroup="Teen"
case Age≥65
  AgeGroup="Senior"
defaultcase
  AgeGroup="Adult"
endcase
```

This example has included one statement for each **case** statement, but there is no limit to the number of statements that may be included in each section. However, there is a maximum limit of 75 **case** statements per **endcase** statement.

LOOP Statements

A **loop** allows Panorama to repeat a sequence of statements over and over again. The loop can be repeated a fixed number of times, or until a special condition is fulfilled.

All loops begin with the **loop** statement, and end with either **until** or **while**. The statements in between these two statements are said to be "inside the loop." These are the statements that will be repeated over and over again. Although it is not required, your procedures will usually be easier to read and understand if the statements inside the loop are indented.

To repeat the statements inside the **loop** a fixed number of times, use the **until** statement with a number after it. This number may be a fixed number, or a variable or formula that calculates a number. For example, this procedure will add a dozen shiny new records to the database:

```
loop
  addrecord
until 12
```

To repeat the statements inside a loop until a specific condition is met, put a true/false formula after the **until** statement. Like the previous example, this example adds new records to the database. In this case, however, the number of new records is determined by asking the user (with the **gettext** statement, see “**GETTEXT**” on page 5317 of the *Panorama Reference*).

```
local NewCount
NewCount="1"
gettext "How many new records?",NewCount
NewCount=val(NewCount)
loop
  NewCount=NewCount-1
  AddRecord
until NewCount=0
```

The **while** statement is the exact opposite of the **until** statement; it repeats the loop as long as the formula remains true. Here is the previous example rewritten to use the **while** statement.

```
local NewCount
NewCount="1"
gettext "How many new records?",NewCount
NewCount=val(NewCount)
loop
  NewCount=NewCount-1
  AddRecord
while NewCount>0
```

These two examples are exactly the same except for the last line.

Note: The **while** statement can also be followed by the special word **forever**, which tells Panorama to repeat the loop forever. Usually this is used with a **stoploopif** statement to break the loop, otherwise your procedure won't ever stop!

Stopping a Loop in the Middle

The **stoploopif** statement allows Panorama to break out of the loop in the middle (or even at the top), instead of at the bottom. Panorama will break out of the loop if the true-false formula is true.

The example below finds every record where the field **PrintDuplicate** contains **Yes**. Each of these records is duplicated. But what if there were no such record? The **stoploopif** statement will stop the loop before it ever begins. The **stoploopif** statement also checks each time the loop is repeated to see if the next statement has found another record to duplicate, or if the loop is done.

```
toprecord
find PrintDuplicate="Yes"
loop
  stoploopif info("notfound")
  copyrecord
  pasterecord
  downrecord
  next
while forever
```

Notice that this sample uses **while forever**. This means that the **while** statement will never stop the loop.

Restarting a Loop in the Middle

The `repeatloopif` statement tells Panorama to restart the loop from the top. The example procedure below tries to extract a phone number from the clipboard.

```
local X,theChar,aPhone
X=1
aPhone=""
loop
  theChar=clipboard()[X;1]
  X=X+1
  stoploopif theChar=""
  repeatloopif theChar≠ "(" and aPhone=""
  aPhone=aPhone+theChar
until aPhone match "(???) ???-????"
```

Each time the loop goes around it copies the next character from the clipboard into the variable `theChar`. If there are no more characters, the loop stops. Each character is checked to see if it is a left parenthesis. Until a (is found, the `repeatloopif` statement stops the loop short, repeating only the top portion of the loop. Once the (is found the loop starts collecting the following data into `aPhone`. The loop finally stops when the entire phone number is collected or the clipboard runs out of data.

Subroutines

Sometimes you may need to use the exact same series of steps in several places in your program. Wouldn't it be nice if Panorama had a special statement that performed this series of steps for you, so you wouldn't have to type those same steps over and over again? Your programs would be smaller, easier to create, and easier to modify. You can't create your own statements, but a subroutine is the next best thing.

A subroutine is used by "calling" it. It's sort of like calling someone to dinner. When a subroutine is called, Panorama temporarily stops performing the steps in the current procedure. It marks its place in the current procedure, and then starts performing the steps in the subroutine. When it has completed all the steps in the subroutine Panorama goes back to the original procedure and starts off right where it left off. The net effect is as if the statements from the subroutine were copied into the middle of the original procedure.

When the same steps are used in different places, a subroutine has many advantages. First of all, using a subroutine makes the database smaller, because these statements appear only once. An even bigger advantage is that if the statements in the subroutine ever need to be changed, they will only have to be changed in one place, instead of over and over again.

It's possible for the main procedure and the subroutine to pass values back and forth between them. These are called parameters. Parameters allow very general subroutines to be written that can handle a wide variety of situations.

CALL Statement

The call statement allows any procedure in the current database to be called as a subroutine. The basic format is simple:

```
call <procedure name>
```

For example, suppose you have created a procedure called `DuplicateRecord`. The procedure looks like this:



We can use this procedure in another procedure by calling it.

```
toprecord
find PrintDuplicate="Yes"
loop
  stoploopif info("notfound")
  call DuplicateRecord
  next
while forever
```

Each time Panorama repeats the loop it will call the [DuplicateRecord](#) procedure. The three steps in that procedure will be performed, then it will return to the loop and perform the next statement. As far as Panorama is concerned, this is exactly the same as if you had written the procedure this way.

```
toprecord
find PrintDuplicate="Yes"
loop
  stoploopif info("notfound")
  copyrecord
  pasterecord
  downrecord
  next
while forever
```

Although there is no difference as far as running the procedure is concerned, there is a big difference for writing procedures. Suppose you have many procedures that need to duplicate a record. If you create a subroutine to duplicate the record, you can save two lines of typing each time you need to duplicate a record. Most subroutines have more than three lines, so the savings are even more substantial.

An even more important advantage is that using subroutines allows you to “modularize” your code. You’ll probably never have to modify the simple code needed to duplicate a record, but more complicated subroutines often need to be adjusted from time to time. If you had simply typed the statements of the subroutine into each location where they were needed (as shown in orange above) then making the adjustment would be very time consuming because you would have to locate and modify each copy of the statements. By collecting these statements together in a subroutine you can make any adjustments necessary to the code in a single location. Every procedure that calls the subroutine will automatically get the benefit of the adjustments.

Calling Procedures With Unusual Names

If a procedure has a space or other punctuation inside the procedure name, you must enclose the procedure name in quotes, like this:

```
call "Calculate P/E Ratio"
```

Quotes are not necessary for a procedure name that contains a period, even if the period is the first character of the name.

```
call .DialNumber
```

It is even possible to calculate the procedure name with a formula. The formula must be surrounded with parentheses. The example below assumes that there is a dialing procedure for several different fields in the database, [Dial Name](#), [Dial Company](#), etc. If there is such a procedure for the current field, this procedure will call it.

```
call ("Dial "+info("fieldname"))
if error
  message "Sorry, can't dial the "+info("fieldname")
endif
```

If there is no such procedure, the error message will appear.

Passing Values to a Subroutine (Parameters)

There are a couple of ways to communicate values between the original procedure and the subroutine. One is to simply put the values in one or more fileglobal or global variables.

A more flexible method is to use **procedure parameters**. A subroutine may have one or more procedure parameters. Each procedure parameter is numbered, starting from 1. When you call the subroutine, you must pass the procedure parameters after the subroutine name. Each parameter must be separated from the next with a comma, like this:

```
call <procedure>,<parameter 1>,<parameter 2>, ...
```

Each procedure parameter may be a field, a variable, a text or numeric constant, or a complete formula. However, if you are going to change a parameter with the **setparameter** statement, that parameter must be a field or a variable.

Inside the procedure, the programmer can use the **parameter()** function to retrieve the parameter values (see "**PARAMETER()**" on page 5592 of the *Panorama Reference*). This function itself has one parameter: the procedure parameter number, for example **parameter(1)**, **parameter(2)**, etc.

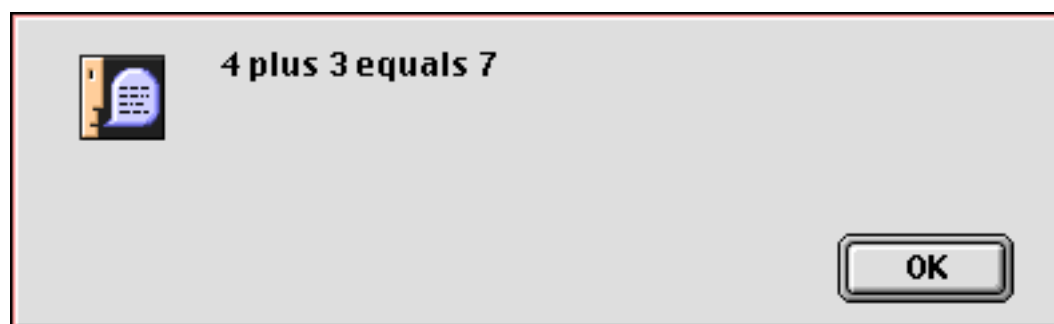
Here is a silly little procedure named **Addition** that displays the result of an addition problem.

```
message str(parameter(1))+ " plus "+str(parameter(2))+ " equals "+
str(parameter(1)+parameter(2))
```

Any other procedure in the same database can call this procedure with two numeric parameters, like this.

```
call Addition,4,3
```

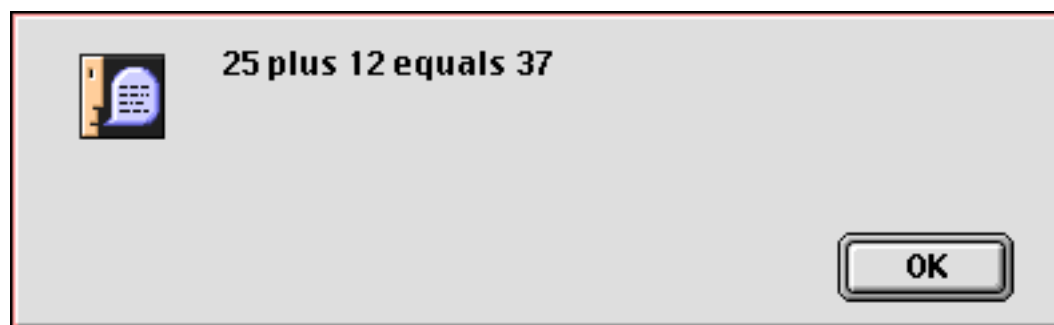
When you run this procedure it calls the subroutine and displays this alert.



By changing the parameters you can change the result.

```
call Addition,35,12
```

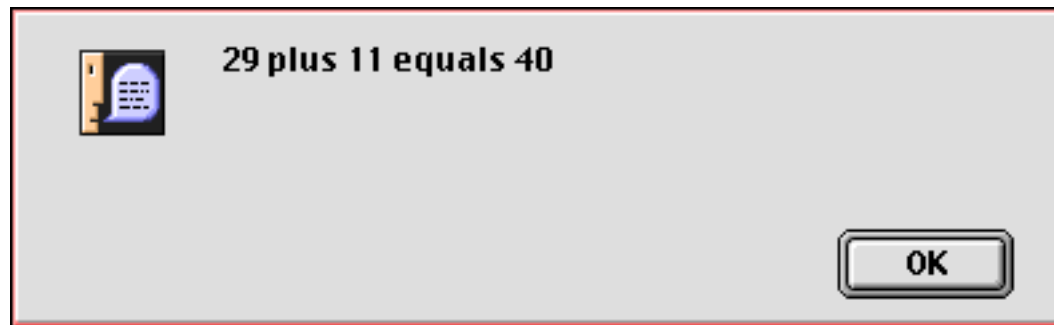
The subroutine grabs the parameters and puts up the result.



Each parameter may be a complete formula containing variables, constants, operators and functions.

```
call Addition,4*2+3*7,sqr(121)
```

Panorama will compute each parameter and pass it to the subroutine.



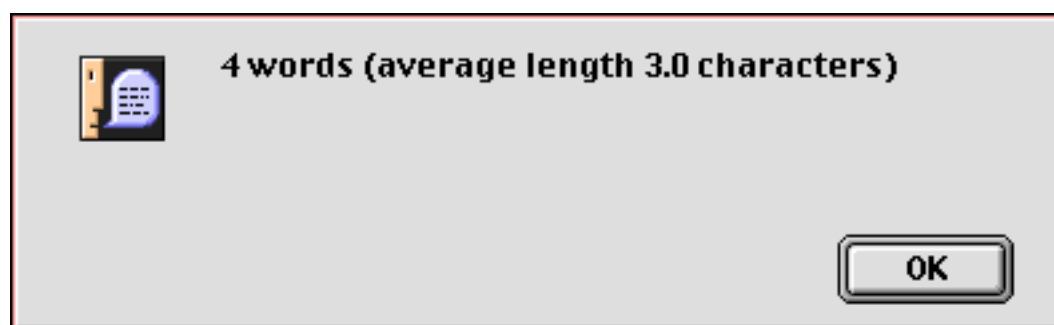
Subroutine parameters can be numbers or text. Here is a procedure named `WordStats` that takes a single text parameter.

```
local words,wordcount,letters
words=parameter(1)
wordcount=arraysize(words," ")
letters=stripchar(words,"AZaz")
message str(wordcount)+" words (average length "+
        pattern(length(letters)/wordcount,"#.#")+ " characters)"
```

This procedure can be called as a subroutine like this.

```
call WordStats,"Now is the time"
```

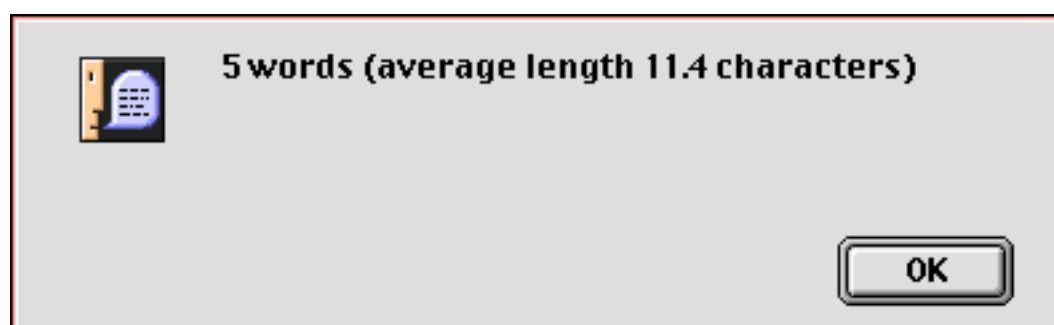
Here is the result.



Just as with the previous example you can pass any data you want to this subroutine.

```
call WordStats,"Dysfunctional institutions instantiate excessive gobbledigook"
```

The subroutine calculates the new statistics.



Passing Values Back From a Procedure

The subroutine can also change a parameter value using the `setparameter` statement (see “[SETPARAMETER](#)” on page 5747 of the *Panorama Handbook*). This statement itself has two parameters, the procedure parameter number you want to change, and new value.

```
setparameter <number>,<value>
```


Here's a procedure named `Weekend` that decides whether the current day is during the week or on a weekend.

```
if datapattern(today(),"DayOfWeek") beginswith "S"
  setparameter 1,"Weekend"
else
  setparameter 1,"Weekday"
endif
```

Any other procedure in this database can call this procedure to find out if today is a weekday or a weekend. This procedure adds a new record to the database on weekdays but not on weekends. The parameter is the local variable `TypeOfDay`.

```
local TypeOfDay
call Weekend,TypeOfDay
if TypeOfDay="WeekDay"
  addrecord
endif
```

A parameter can be passed to a procedure and then back again. Here's a modified version of the `Weekend` procedure that works for any day, not just today.

```
if datapattern(parameter(1),"DayOfWeek") beginswith "S"
  setparameter 1,"Weekend"
else
  setparameter 1,"Weekday"
endif
```

Here is a procedure that uses this revised subroutine. Notice that the `DayInfo` variable is assigned a value (`December 7, 1941`) before being passed to the procedure. The procedure gives the `DayInfo` variable a new value (`Weekend`).

```
local DayInfo
DayInfo=date("December 7, 1941")
call Weekend,DayInfo
message "Pearl Harbor was bombed on a "+DayInfo
```

An important point to understand is that the subroutine does not know the name of the field or variable it is modifying. It could be `DayInfo`, `TypeOfDay`, or `ZippityDoo` — it's up to the procedure that calls the subroutine.

In the previous section you learned that a parameter can be any formula, for example `3*4` or `array(Address,2,¶)`. However, this is not true for parameters that are modified by the subroutine. A parameter that is going to be modified must be a field or variable. For example, you cannot call the `Weekend` subroutine like this.

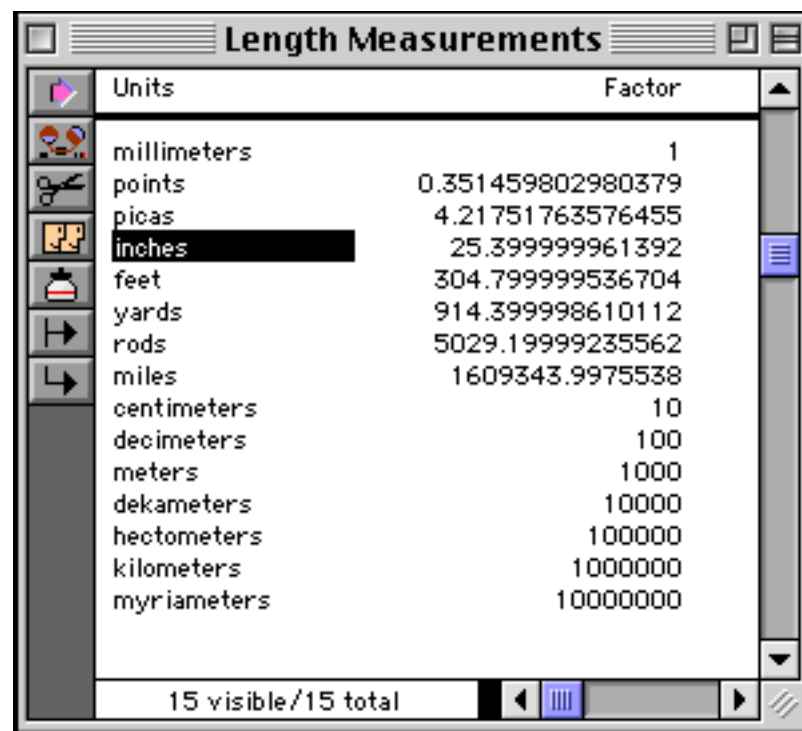
```
local DayInfo
call Weekend,date("December 7, 1941") <--- WRONG
message "Pearl Harbor was bombed on a "+DayInfo
```

The problem with this procedure is that the subroutine has no idea where to put the result.

Here is a more useful subroutine that uses parameters. This subroutine (named `ConvertLength`) can convert one measurement system into another (for example feet into meters).

```
/* call ConvertLength,dimension,from,to */
local from,to,length
from=lookup(info("databasename"),Units,lower(parameter(2)),Factor,0,0)
to=lookup(info("databasename"),Units,lower(parameter(3)),Factor,0,0)
if from=0 or to=0 rtn endif
length=(parameter(1)*from)/to
setparameter 1,length
```

The `ConvertLength` subroutine is designed to be part of this database, which contains the measurement factors used by the `lookup()` functions in the procedure (see above).

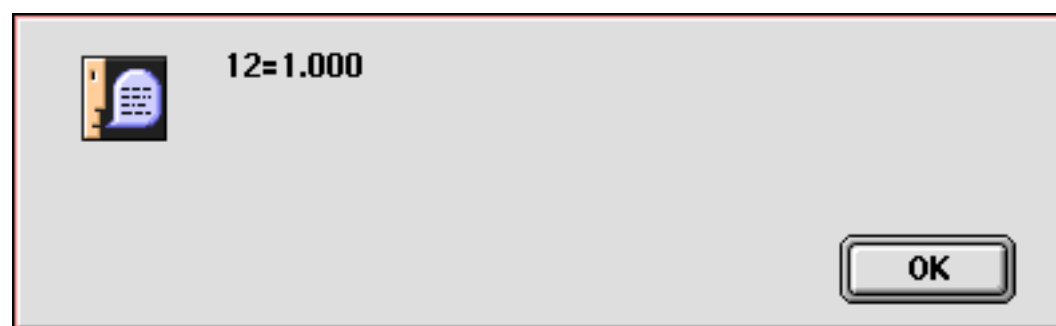


Units	Factor
millimeters	1
points	0.351459802980379
picas	4.21751763576455
inches	25.399999961392
feet	304.799999536704
yards	914.399998610112
rods	5029.19999235562
miles	1609343.9975538
centimeters	10
decimeters	100
meters	1000
dekameters	10000
hectometers	100000
kilometers	1000000
myriameters	10000000

Here is a procedure that uses the subroutine to convert 12 inches into feet.

```
local original,converted
original=12
converted=original
call ConvertLength,converted,"inches","feet"
message str(original)+"="+pattern(val(converted),".###")
```

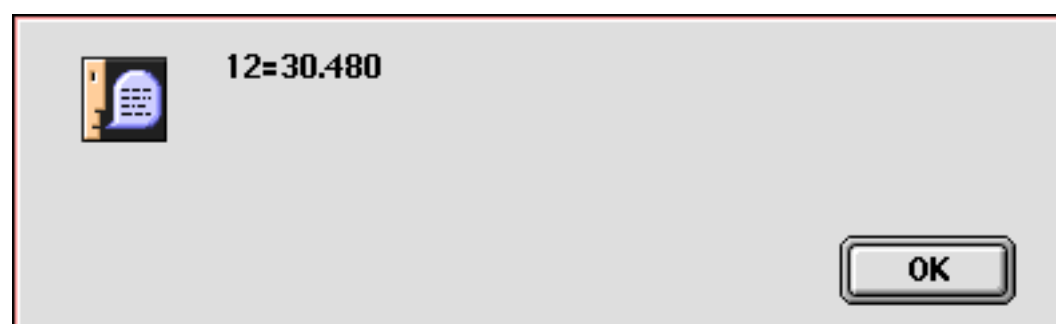
The result is this alert.



By making a slight adjustment we can convert 12 inches into centimeters.

```
local original,converted
original=12
converted=original
call ConvertLength,converted,"inches","centimeters"
message str(original)+"="+pattern(val(converted),".###")
```

The result is this alert. News flash — 12 inches equals 1 foot.



This procedure/subroutine combination illustrates a quirk in the way Panorama handles numeric parameters. If you look at the original procedure you will notice that it is setting the parameter to a numeric value.

```
/* call ConvertLength,dimension,from,to */
local from,to,length
from=lookup(info("databasename"),Units,lower(parameter(2)),Factor,0,0)
to=lookup(info("databasename"),Units,lower(parameter(3)),Factor,0,0)
if from=0 or to=0 rtn endif
length=(parameter(1)*from)/to
setparameter 1,length
```

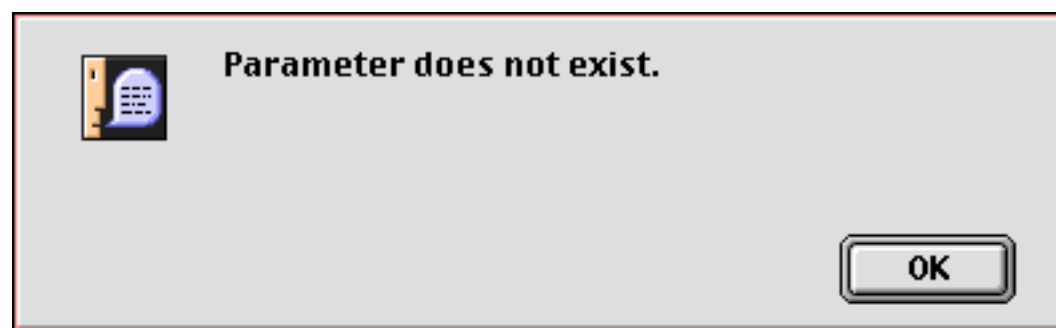
However, the **setparameter** statement always converts numbers to text when it stores the result in a variable. Because of this the calling procedure must use the **val()** function to convert the number back into a number again.

```
local original,converted
original=12
converted=original
call ConvertLength,converted,"inches","centimeters"
message str(original)+"="+pattern(val(converted),"#.###")
```

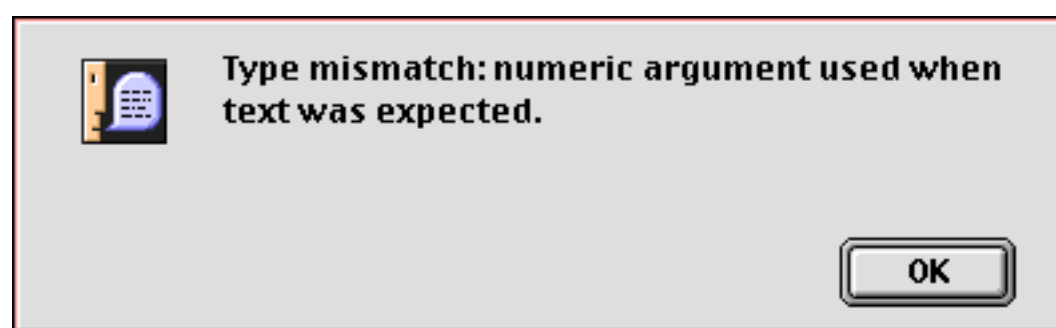
You'll need to keep this in mind if your subroutine passes back numeric values. (Note: This quirk could probably be considered a bug. However, if we fixed it then all of the databases that currently work (by using the **val()** function) would suddenly become broken. Therefore we intend to keep it this way.)

What if the parameters don't match the procedure?

Like a hand in a glove, the procedure parameters supplied as part of the **call** statement must exactly match the parameters used by the procedure being called. For example, consider the **ConvertLength** procedure in the last example. If you call this procedure, you must supply at least three parameters. (It's ok to supply more than three...the extra parameters will be ignored.) If you supply less than three parameters, the **ConvertLength** procedure will stop and an error message will be displayed when it tries to access a missing parameter.



In addition to having the correct number of parameters, the parameters must also have the correct data type. In our **ConvertLength** example, the first parameter supplied must be a number, while the second and third parameters must be text. If the wrong type of data is passed in the parameter, the procedure will stop and display an error message when you try to use the value. Here's the message that appears if a numeric parameter is passed when a text parameter is required, a similar message appears for the opposite case.



It's possible to check for missing parameters in a procedure using the **if error** statement. This allows you to perform your own action instead of displaying the default error message. Here's a revised version of the **ConvertLength** procedure that displays custom error messages and then stops if a parameter is missing.

```

/* call ConvertLength,dimension,from,to */
local from,to,length,fromUnits,toUnits
fromUnits=parameter(2)
if error
  message "From units must be inches, feet, yards, centimeters, etc."
  stop
endif
toUnits=parameter(3)
if error
  message "To units must be inches, feet, yards, centimeters, etc."
  stop
endif
from=lookup(info("databasename"),Units,lower(fromUnits),Factor,0,0)
to=lookup(info("databasename"),Units,lower(toUnits),Factor,0,0)
if from=0 or to=0 rtn endif
length=(parameter(1)*from)/to
setparameter 1,length

```

Although this example stops if there is a parameter error, that is not absolutely necessary. If you can determine a reasonable default value for a missing parameter the procedure can simply substitute that value and continue on its way. Here is another variation of the **ConvertLength** procedure that defaults to inches if the a parameter is missing.

```

/* call ConvertLength,dimension,from,to */
local from,to,length,fromUnits,toUnits
fromUnits=parameter(2)
if error
  fromUnits="inches"
endif
toUnits=parameter(3)
if error
  toUnits="inches"
endif
from=lookup(info("databasename"),Units,lower(fromUnits),Factor,0,0)
to=lookup(info("databasename"),Units,lower(toUnits),Factor,0,0)
if from=0 or to=0 rtn endif
length=(parameter(1)*from)/to
setparameter 1,length

```

Here is a procedure that uses the revised subroutine to convert 12 centimeters into inches.

```
local original,converted
original=12
converted=original
call ConvertLength,converted,"centimeters"  <-- missing parameter defaults to inches
message str(original)+"="+pattern(val(converted),"#.###")
```

Or the missing parameter can be in the middle of the list like this. In this case 12 inches will be converted into centimeters.

```
local original,converted
original=12
converted=original
call ConvertLength,converted,, "centimeters"  <-- missing parameter defaults to inches
message str(original)+"="+pattern(val(converted),"#.###")
```

Calling a Subroutine in Another Database

The **call** statement calls another procedure in the current database as a subroutine. (The current database is the database associated with the topmost window — not necessarily the database the current procedure belongs to.) The **farcall** statement can call any procedure in any open database, not just the current one. The format of this statement is almost identical to the **call** statement, but you must specify the database name.

```
farcall <database>,<procedure>,<parameter 1>,<parameter 2>, ...
```

The database name should usually be in quotes, like this.

```
farcall "Length Measurements",ConvertLength,converted,"feet","miles"
```

It's also possible to use a formula to calculate the database name. The example below searches for any open database with the word **Phone** in the name, then attempts to call the **.Dial** procedure in that database.

```
local X,dbList
dbList=info("files")
X=search(dbList,"Phone")
if X=0
  message "No phone database open!"
  stop
endif
X=arrayelement(dbList,X,1)
farcall (array(dbList,X,1)),.Dial,Name
```

When the database name is calculated as in this example, the formula must be surrounded by parentheses ().

Terminating a Subroutine in the Middle

The **rtn** statement (short for **return**) allows a subroutine to stop short in the middle and return to the original procedure. Usually when a subroutine is called, all the statements in the subroutine are performed from top to bottom. But if the **rtn** statement is encountered, the subroutine stops and immediately goes back to the original procedure. The **rtn** statement is almost always used in combination with the **if** or **case** statement.

The simple example below dials a local phone number, which is passed to the subroutine in parameter 1. If no phone number is passed, or if a long distance phone number is passed, the subroutine returns without doing anything.

```
local DialNumber
DialNumber=parameter(1)
if error
    rtn
endif
if DialNumber="" or length(DialNumber)>8
    rtn
endif
dial DialNumber
```

If the **rtn** statement is encountered in a procedure that has not been called as a subroutine (i.e. an original procedure) the procedure will simply stop.

Panorama also has another statement that terminates a subroutine in the middle: **rtnerror**. This statement allows a subroutine to generate an error (complete with error message). If the statement following the original call statement is **if error**, the subroutine will return and the error will be processed by the **if error** statement (see “[Error Handling with if error](#)” on page 258). If there is no **if error** statement, the program will simply stop immediately and display an error message.

Mini Subroutines within a Procedure

Sometimes you may want to use a short subroutine, perhaps two or three lines. It just seems like too much hassle to create a separate procedure. For these situations, Panorama allows you to create a subroutine right inside the current procedure. This special subroutine within a procedure is called a **short subroutine**.

A short subroutine always begins with a **label**. A label is a unique series of letters and numbers that identifies a location within the procedure. The label may not contain any spaces or punctuation except for . and \$, and must always end with a colon. The colon is not actually part of the label, it simply identifies the series of letters and numbers as a label, as opposed to a field or variable. Here are some examples of labels:

```
diamond:
blue:
mailAction7:
Dispatch.Route:
```

A short subroutine ends with the end of the procedure, or with a **rtn** statement. A single procedure may contain many short subroutines, each starting with a label and ending with a **rtn** statement.

Short subroutines are called with the **shortcall** statement. This statement is always followed by the name of the short subroutine (the label). Don't include the colon here. You must type the label exactly as it appears at the top of the short subroutine (except for the colon), no quotes, and unlike a regular call statement the subroutine name cannot be calculated. The **shortcall** statement also does not allow parameter passing. Here are examples of how to call short subroutines.

```
call diamond
call blue
call mailAction7
call Dispatch.Route
```

The example below contains a short subroutine called `GroupTotal`. The short subroutine starts on line 7, with the label `GroupTotal:`. This short subroutine performs three steps and then returns to the main section of the program. The main section of the program calls the subroutine twice, then stops.

```
field City
shortcall GroupTotal
field State
shortcall GroupTotal
stop

GroupTotal:
  groupup
  field "Amount"
  total
  rtn
```

If the `stop` statement was not included, the program would continue down and perform the steps in the short subroutine a third time. In fact, we do this on purpose to produce a shorter version of this program.

```
field City
shortcall GroupTotal
field State

GroupTotal:
  groupup
  field "Amount"
  total
```

We've also removed the `rtn` statement at the end of the short subroutine. It's redundant in this case because the short subroutine ends at the end of the entire procedure. However, if there were additional short subroutines after this one, all but the last one would require a `rtn` statement at the end.

Subroutines and Local Variables

Earlier in this chapter you learned that local variables are destroyed at the end of the procedure that created them (see "[The Birth and Death of a Local Variable](#)" on page 249). Local variables also become dormant when a subroutine is called (except for short subroutines, see "[Mini Subroutines within a Procedure](#)" on page 270). At the end of the subroutine the local variables come out of hibernation. Because of this, local variables created in one procedure cannot be accessed in another procedure. Local variables are always completely separate.

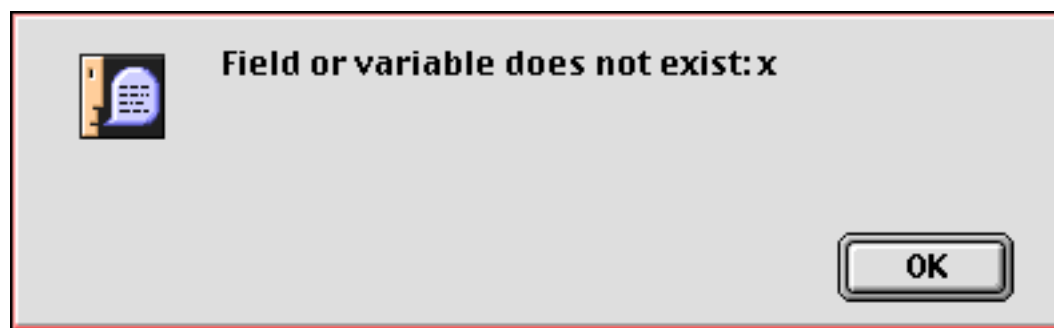
To illustrate this, consider this procedure which creates a local variable named `x`.

```
local x
x=2
call test
message x
```

The procedure `test` contains only one line which displays the `x` variable.

```
message x
```

However, this subroutine does not work! The `x` variable is now dormant and is not accessible to the `test` procedure. Instead of displaying the value `2`, this error message appears.



As a matter of fact, Panorama allows the `test` subroutine to have its own separate `x` variable, like this.

```
local x
x=9
message x
```

Now when the original procedure (see above) is run two alerts appear. The first displays the value `9` (the value of `x` set in the `test` subroutine). The second displays the value `2` (the original value of `x` which was dormant but re-appeared when the `test` subroutine was finished).

The UseCallersLocalVariables and UseMyLocalVariables Statements

As described in the previous section, local variables created in a procedure can normally only be accessed in that procedure and not any other. In some special cases, however, it can be useful for a subroutine to access the local variables of the subroutine that called it. This is especially true when you are creating procedures for use as custom statements, and when using the `execute` statement.

Panorama has two statements that allow a procedure to access the local variables of the procedure that called the current procedure: `UseCallersLocalVariables` and `UseMyLocalVariables`. These statements **must** be used as a pair. (Be careful not to use the `Rtn` statement without first using the `UseMyLocalVariables` statement.) The `UseCallersLocalVariables` statement temporarily swaps out a procedure's current local variables with the local variables of the procedure that called this procedure. The `UseMyLocalVariables` statement swaps the variables back again. While the variables are swapped you have full access to the local variables of the caller. You can even create new local variables, which become local variables belonging to the calling procedure. Here is an example that creates a local variable named `alphabet` and initializes it with 26 letters.

```
usecallerslocalvariables
local alphabet
alphabet="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
usemylocalvariables
```

A procedure could call this subroutine like this. The message statement will display `ABCDEFGHIJKLMNOPQRSTUVWXYZ` even though this local variable wasn't created in this procedure.

```
call MakeAlphabet
message alphabet
```


Recursive Subroutines

A **recursive subroutine** is a subroutine that calls itself. Some programming languages don't allow recursion, but Panorama does allow recursion up to 8 levels deep. Here is an example of a recursive procedure called **AddAddAdd**. Notice that it calls itself on line 4.

```
local x,y
x=parameter(1)
if x>1
    call AddAddAdd,x-1,y
    y=val(y)+x
else
    y=x
endif
setparameter 2,y
```

Given an integer, this procedure will calculate the sum of that integer plus all lower integers. For example, if you start with **4** the procedure will compute $4+3+2+1 = 10$. The procedure calls itself for each addition. Here is a procedure that calls **AddAddAdd** to start the computation. In this case the computation starts with **6**, so the result will be $6+5+4+3+2+1 = 21$.

```
local answer
call AddAddAdd,6,answer
message answer
```

Although Panorama allows recursion it is very limited. The **AddAddAdd** procedure won't work with any number larger than **8**. Usually it's best to try to write a procedure without recursion. Here's an example of a procedure that solves the same problem without recursion. Now we can calculate that for a larger number like **68** our cumulative sum is **2346**.

```
local x,sum
x=68
sum=0
loop
    sum=sum+x
    x=x-1
while x>0
message sum
```

As you can see, the non-recursive solution is actually quite a bit simpler than the recursive solution. However, there are some cases where recursion can greatly simplify the solution of a problem, and Panorama does have a limited capability to allow recursion.

Using a Subroutine in a Formula (the CALL(function)

Subroutines are usually called from other procedures, but the **call(** function allows a subroutine to be called from inside a formula. This means that a procedure can be used anywhere you can use a formula — to display data on forms and reports, in a Live Menu definition, in a formula fill, anywhere.

The **call(** function has two required parameters and may also include any number of additional parameters.

```
call(database,procedure,parameter1,parameter2,parameter3, ... )
```

The first parameter, **database**, is the name of the database that contains the procedure to be called. If this procedure is in the same database you can simply use **"**.

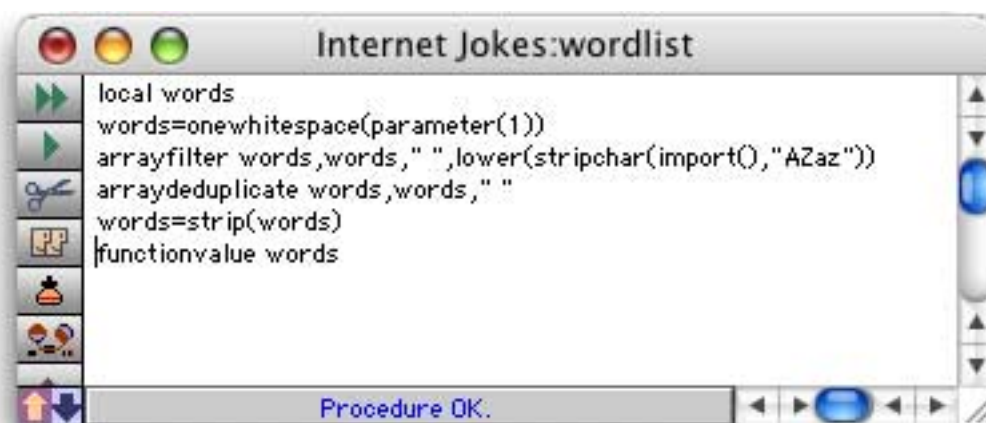
The second parameter, **procedure**, is the name of the procedure to be called. Usually the procedure name must be quoted, for example **"CubeRoot"**.

If there are any additional parameters their values are passed along to the procedure, where they can be accessed with the `parameter()` function (see “[Passing Values to a Subroutine \(Parameters\)](#)” on page 263). The procedure can also find out how many parameters were passed by using the `info("parameters")` function. (Note: Unlike a regular procedure you cannot pass a value back to a parameter with the `setparameter` statement.)

A procedure called within a formula usually a value. This is done with the `functionvalue` statement, which has one parameter — the value to be returned. This may be either a text or numeric value. (If the procedure does not execute a `functionvalue` statement the default value "" will be returned.)

```
functionvalue value
```

Lets look at an actual example. Here is a procedure designed to take arbitrary text and turn it into an alphabetized list of words contained within the original text.



This procedure can be called in the **Formula Fill** command by using the `call()` function. In this case the text to be converted into a list of words is in the `Joke` field.



It takes a bit longer to run the **Formula Fill** command than usual, but when it's done you'll see that it has compiled an alphabetical list of words contained in each joke.

Title	Rating	Joke	Words
Household Principles for Children	G	Household Principles for Children Based on the	a abomination about absolutely acceptable affli
Dinner with God	G	Dinner with God.	after all am an and announced announcements b
Planet of the Chain Letters...	G	Please distribute this to everyone you know or	ape bury days distribute dress earth everybod
Techie joke	G	Remember the millions of mutations of the stri	a actually after ago and bar bartender be beer
Old aviation jokes	G	The controller working a busy pattern told the	a able after ahead airplane airport already alw
Feeling stressed?	PG	Picture yourself near a stream...	a air are better birds bother called can cascadi
Computer age haiku	PG	Gentle words for our age :	a aborted absence age all am and are ask be be.
MICROSOFT Bids to Acquire Cathc	PG	By Hank Vorjes VATICAN CITY (AP) -- In a joi	a abrahamic absolutioneven access according a
Nice Performance report	PG	Subject: Nice Performance report	a about absolutely accomplishments always an
Important Warnings	G	As scientists and concerned citizens, we appla	a about above according actuality adhesive adv
Steven Wright Wrocks	PG	Last night I played a blank tape at full blast. Th	a about above after amphibians an and anyhow
THE DIFFERENCE BETWEEN HYMNS	PG	An old farmer went to the city one weekend an	a an and are asked attended be big black brown
Tech Support Joke	PG	A cannibal goes to the human meat market to b	a about an analysts and any are as asks assista
The Winner...	PG	A woman gets home.....	a and at back bags beach car doesnt door drive
Hospital Bill	G	A man was brought to Mercy Hospital, and tak	a afraid and any are as be bed bill brotherinlaw
Spiders	PG	Little Lucy was playing in the garden when she	a and are asked both call daddy do doing flat fo
The sea captain.	G	One of the world's most famous sea captains d	a about admired after aloud an and astonished :

This function can also be used in a form to supply data for a Text Display, Text Editor, Auto-Wrap Text, or Super Flash Art object. In this example we'll use a Text Editor SuperObject.

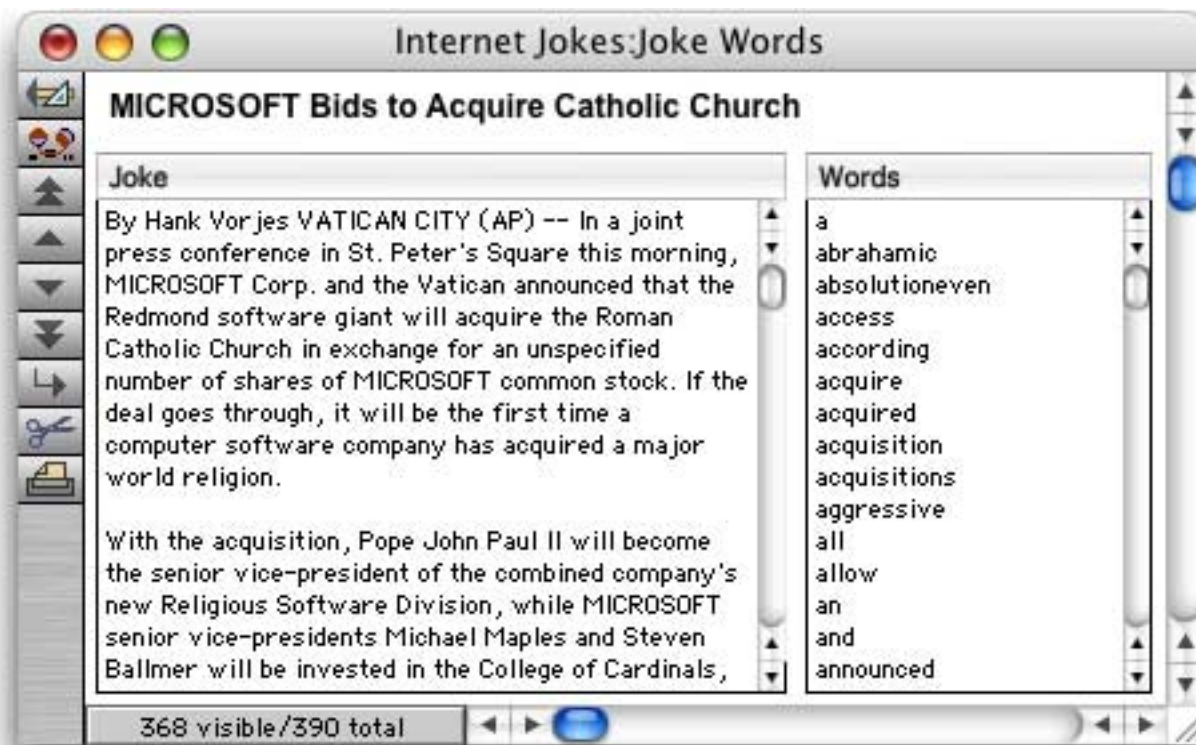
Text Editor SuperObject

Data

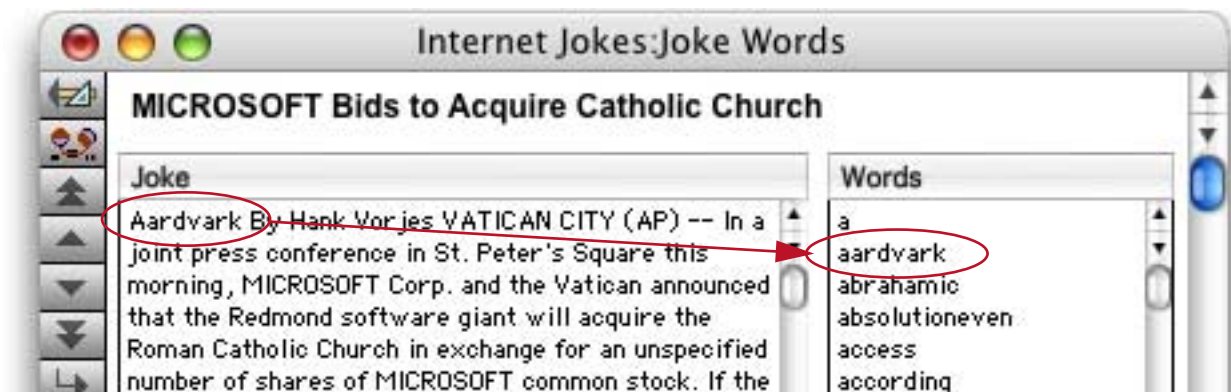
Field
 Variable
 Formula

replace(call("", "wordlist", Joke), " ", cr())

Here is the form that contains this object. The Text Editor object that uses this formula is on the right. (Notice that the Words field is not reference in this formula — it is not needed and can be left out of the database. The list of words is calculated on the fly as the form is displayed.)



If a new word is added to the joke it will automatically be displayed in the list of words.



Of course you can also use the `call()` function within a procedure, like this.

```
local mywords
mywords=call("", "wordlist", Joke)
message str(linecount(mywords))+" different words were used in this joke."
```

The `call()` function brings the power of procedures to any formula.

Restrictions on Subroutines used as Formulas. There are some significant restrictions that any procedure called by the `call()` function must follow:

The procedure cannot have any database related side effects. This means that it cannot modify any database fields (variables are ok). It cannot move up or down (or search) to a different record in the current database — it must stay on the current record. It cannot move left or right to a different field.

The procedure cannot switch to a different window (this includes secret windows) or switch to a different databases.

The procedure can't change the view in the current window (for example it cannot change from data sheet to form or change from data mode to graphics mode).

The procedure cannot create a permanent variable (other types are ok, and it is ok to change the value of a permanent variable that already exists).

The procedure cannot display a dialog based on a form. You can display an alert, though this is almost always a bad idea and can cause endless loops (for example if an alert appears when a form object is displayed).

For the most part it's up to you to follow these rules. If you violate these rules Panorama will stop the procedure and the formula with the `call()` function will fail to calculate a value. (In some situations Panorama may even crash so take care if you are pushing the envelope.)

You should also be careful to make the procedure run as fast as possible. Since the procedure may be triggered by a formula in a form that is displayed or printed, or in a `formulafill` statement, it should be kept as short and fast as possible.

Other Control Flow Statements

There are a few other control flow statements that don't fit into any neat categories. These statements are described in the following sections.

Jumping to an Another Location in the Program

The `goto` statement tells Panorama to jump immediately to another spot in the program (see "[GOTO](#)" on page 5326 of the *Panorama Reference*). The format of this command is simple:

```
goto <label>
```

A **label** is a unique series of letters and numbers that identifies a location within the procedure. The label may not contain any spaces or punctuation except for `.` and `$`, and must always end with a colon. The colon is not actually part of the label, it simply identifies the series of letters and numbers as a label, as opposed to a field or variable. Here are some examples of labels:

```
diamond:
```

```
tryAgain:
```

```
accessRoute3:
```

```
Start.Over:
```

The example subroutine below asks the user to enter an angle. If the angle is not between 0 and 360, Panorama will jump back to the label `TryAgain`.

```
fileglobal GoAngle
GoAngle="0"

TryAgain:
  gettext "Enter direction (0-360)",GoAngle
  if GoAngle>360 or GoAngle<0
    goto TryAgain
  endif
  setparameter 1,val(GoAngle)
```

You may wonder why the `goto` statement is buried here at the end of the chapter. It's simple: we don't want you to use it! At least, not much. For years professional programmers have known that too many `goto`'s quickly create "spaghetti" code that is usually confusing. In fact, it has been mathematically proven that any program can be written with `if` and `loop` statements, without any `goto`'s at all. Here's how our example could be rewritten without a `goto`.

```
global GoAngle
GoAngle="0"
loop
  gettext "Enter direction (0-360)",GoAngle
while GoAngle>360 or GoAngle<0
  setparameter 1,val(GoAngle)
```

As you can see, this example is actually simpler without the `goto` statement. Occasionally the `goto` statement does make it easier to write a program. Frankly we couldn't come up with an example of this, but if you do, feel free to use the `goto` statement. That's what it's there for.

Stopping the Program

The `stop` statement tells Panorama to stop the procedure immediately (see "[STOP](#)" on page 5796 of the *Panorama Reference*). If the current procedure is a subroutine, the original procedure is also stopped. If you want the current subroutine to stop but the original procedure to continue, use the `rtn` statement (see "[Terminating a Subroutine in the Middle](#)" on page 269). The `rtn` statement also acts to stop the procedure if it is not being used as a subroutine.

Aborting a Program

Sometimes you may need to stop a program in the middle, before it has finished running. For example, suppose you make a mistake and create a loop that never stops. If that happens you need to abort the program. On the Macintosh you can do this by pressing **Command-Period**, on the PC by pressing **Control-Period**.

The ability to abort any program is normally an important safety valve, but you may have a procedure that should not be stopped in the middle with a job halfway done. For these types of cases Panorama allows you to disable the ability to abort during some or all of a procedure.

To disable the abort feature use the `disableabort` statement. To re-enable the ability to abort use the `enableabort` statement. (If you don't include an `enableabort` statement Panorama automatically re-enables aborting at the end of the procedure.)

The example below shows how these statements can be used. In this case there is no way that the new record can be added without being filled in by the lookup formulas. You either get all or nothing, but not a halfway done job.

```
disableabort
addrecord
Name=dialogName
Address=lookup("Contacts","Name",Name,Address,"",0)
City=lookup("Contacts","Name",Name,City,"",0)
State=lookup("Contacts","Name",Name,State,"",0)
Zip=lookup("Contacts","Name",Name,Zip,"",0)
enableabort
```

When using the **disableabort** statement you must be careful, especially when using loops. The procedure below will hang Panorama. The only way to stop the loop is to reboot the computer or do a force quit on Panorama (**Command-Shift-Option-Escape** on the Macintosh, **Control-Alt-Delete** on the PC).

```
disableabort
local i,tag
i=1
loop
    tag="<"&array(Text,i,¶)&">"
    stoploopif tag=""
    i=i+1
while forever
enableabort
```

While this example may look silly, it is easy to create an endless loop without realizing it.

Controlling the Abort Process

Sometimes you may want to allow a procedure to be aborted before it is finished, but in a controlled way. For example, suppose a procedure opens a progress window then loops over and over again to perform some operation (perhaps copying files or some other slow function). You might want to allow the procedure to be cancelled before it finishes, but you want to make sure that it closes the progress window even if the procedure is aborted. This can be done with the **info("abort")** function. This function returns a true or false value depending on whether the **Command-Period** (Mac), or **Control-Period** (PC) key has been pressed. Here is an example of a procedure that stops the loop if these keys are pressed.

```
disableabort
loop
    if info("abort")
        alert 1014,"Abort?"
        if info("dialogtrigger") contains "yes"
            stoploopif 1=1
        endif
    endif
    ...
    ... body of loop
    ...
while forever
    ...
    ... clean up after loop (close temporary windows, etc.)
    ...
enableabort
```

Since the procedure itself is testing to see if it should abort it is able to abort cleanly, finishing up any necessary tasks like closing progress windows, etc. This is much better than simply stopping the procedure at a semi-random spot.

Important note: The `info("abort")` function will only return true ONCE for each time the **Command-Period** (Mac), or **Control-Period** (PC) key combination is pressed. If you need to test it more than once (for example to cancel two nested loops you must copy the result into a variable and then test the variable. In other words, you generally don't want to have the `info("abort")` function in more than one spot within a loop.

Doing Nothing for a While

The `nop` statement (short for no operation) tells Panorama to do absolutely nothing! You can use the `nop` statement as a placeholder, or to delay for a short time. Here's an example of a procedure that will delay for a short time (probably less than a second).

```
loop
  nop
until 20
```

The exact amount of time delay depends on the speed of your computer. Here's an example that will delay exactly 10 seconds.

```
local startTime
startTime=now()
loop
  nop
until now(>startTime+10
```

Another use for the `nop` statement is to fool Panorama into not displaying a warning dialog. When used as the last statement in a procedure, or just before a `stop` statement, statements like `quit` and `close` will ask the user if they want to save changes. By adding a `nop` statement you can prevent this dialog from appearing.

```
if info("trigger") contains "Close w/o Save"
  close
  nop
  stop
endif
```

See “**NOP**” on page 5545 of the *Panorama Reference* to learn more about doing nothing with the `nop` statement (just kidding, there's nothing more to learn!).

Building Subroutines On The Fly (The Execute Statement)

Subroutines are usually written in advance using the procedure editing window (see “[Writing a Procedure from Scratch](#)” on page 216). However it is possible for a procedure to construct a subroutine “on-the-fly” and then immediately call (execute) that subroutine. This magical trick is performed by a special statement called `execute`. The `execute` statement is followed by a formula that calculates the text of the subroutine to be called.

```
execute <formula>
```

Here is an example procedure that uses the `execute` statement to “ditto” the value in the cell above the current cell. Basically, this procedure moves up a line, grabs the cell, then moves down a line and assigns the new value to the current cell.

```
local dittoValue,dittoField
dittoField=info("FieldName")
uprecord
if stopped rtn endif
dittoValue=«»
downrecord
execute dittoField+="{ "+dittoValue+"}”
```


The first line (`local dittoValue,dittoField`) simply sets up the two local variables we will need (see “[Variables](#)” on page 247).

The second line (`dittoField=info("FieldName")`) gets the name of the current field and places it in the `dittoField` variable. In the example above this value will be `Company`.

First	Last	Credit Card	Title	Company	Address
Jim	Nickle		President	Jim's Appliances	14189 8th
Logan	Nourse		Purchasing	Palo Alto Lumber	1828 Amaranta
Sam	Pack				6051 Pheasant
Michael	Paine				625 S.E. High

The third line (`uprecord`) moves up one record, like this.

First	Last	Credit Card	Title	Company	Address
Jim	Nickle		President	Jim's Appliances	14189 8th
Logan	Nourse		Purchasing	Palo Alto Lumber	1828 Amaranta
Sam	Pack				6051 Pheasant
Michael	Paine				625 S.E. High

The fourth line (`if stopped rtn endif`) checks to see if we were already on the top line of the database, and if so, stops the procedure.

The fifth line (`dittoValue=«»`) grabs the value in the current cell, in this case `Palo Alto Lumber` (see “[Using the Current Field](#)” on page 52).

The sixth line (`downrecord`) moves back down one record to the original position.

First	Last	Credit Card	Title	Company	Address
Jim	Nickle		President	Jim's Appliances	14189 8th
Logan	Nourse		Purchasing	Palo Alto Lumber	1828 Amaranta
Sam	Pack				6051 Pheasant
Michael	Paine				625 S.E. High

The seventh line (`execute dittoField+="{ "+dittoValue+"}"`) assigns the value in `dittoValue` to the cell.

First	Last	Credit Card	Title	Company	Address
Jim	Nickle		President	Jim's Appliances	14189 8th
Logan	Nourse		Purchasing	Palo Alto Lumber	1828 Amaranta
Sam	Pack			Palo Alto Lumber	6051 Pheasant
Michael	Paine				625 S.E. High

Let's take a close look at how the `execute` statement did its job. Here's the formula it used.

```
dittoField+="{ "+dittoValue+"}"
```

Now we know that `dittoField` is `Company`, and `dittoValue` is `Palo Alto Lumber`. So the result of this formula is this —

```
Company={Palo Alto Lumber}
```

Now this is a valid assignment statement (see “[Assignment Statements](#)” on page 243) that assigns a value into the `Company` field! (In case you have forgotten, `{` and `}` are alternate quote characters that can be used instead of `"`. See “[Constants](#)” on page 49 for a complete list of quote characters.) This line is a completely valid subroutine all by itself. So Panorama goes ahead and executes this custom subroutine which causes the company name to be filled in.

If we move one column to the left, the formula will generate a different custom subroutine.

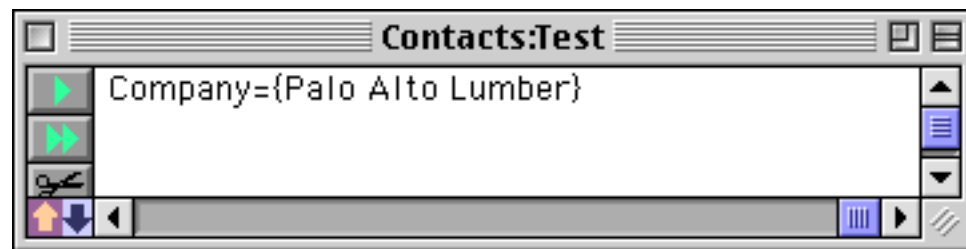
```
Title={Purchasing}
```

The ditto procedure will work for any text field. (It could be revised to work with numeric and date fields as well with some extra work with the `str()` and `datepattern()` functions.)

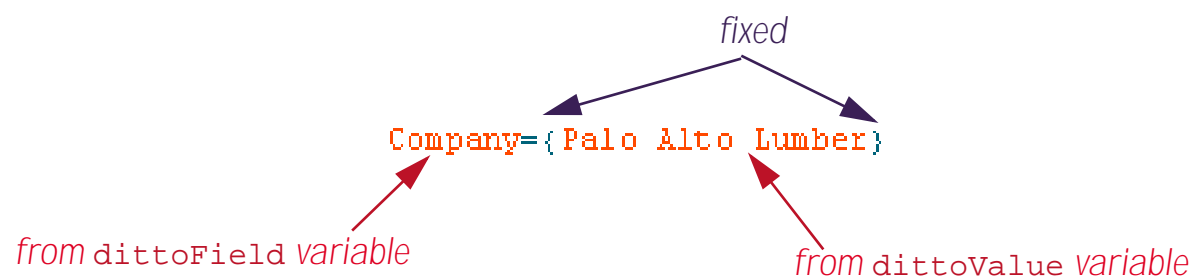
Tips for On-The-Fly Program Writing

Writing a regular program can be tricky enough. Attempting to create a procedure on the fly can turn into a quagmire if you don’t take a systematic approach. Careful planning will insure success.

Before you start attempting to write a formula make sure you have a good idea of what the final subroutine will look like. If the subroutine is more than one or two lines it’s a good idea to create a “mock up” using the procedure editor. This should be a fully running procedure hard coded for specific data. Here’s an example mock up for the ditto cell procedure described in the previous section.



Once the mock up is debugged you can start converting it into a formula. Start by figuring out which sections of the procedure are fixed and which will change, and where the data for the changeable parts will come from.



Now we’ll start to assemble the formula. The first part is changeable, and comes from the `dittoField` variable. So the first part of the formula is simply

```
dittoField
```

Next is a fixed component `={`. We want this to appear exactly like this in the final subroutine, so we quote it and concatenate it to the first section. Since we’re using `{` and `}` for quotes in the final procedure we’ll need to use a different kind of quote here. We’re using `"` here but we also could have used smart quotes (see “[Constants](#)” on page 49 for a list of different types of quotes).

```
dittoField+"="{
```

The next component is changeable, and comes from the `dittoValue` variable, so we’ll add that on next.

```
dittoField+"="{+dittoValue
```

Finally we'll add on the closing } quote. Again this must be enclosed in a different type of quote.

```
dittoField+="{ "+dittoValue+" }"
```

The trickiest part is often the nested quotes. Each type of quote must be nested in another type. An alternative technique would be to use the `chr()` function (see "[CHR](#)" on page 5099 of the *Panorama Reference*) to generate the { and } quotes, like this.

```
dittoField+=" "+chr(123)+dittoValue+chr(125)
```

The values 123 and 125 must be looked up in an ASCII chart. A partial ASCII chart is shown below. (Use the [ASCII Wizard](#) to see complete ASCII chart — see "[The ASCII Chart Wizard](#)" on page 89).

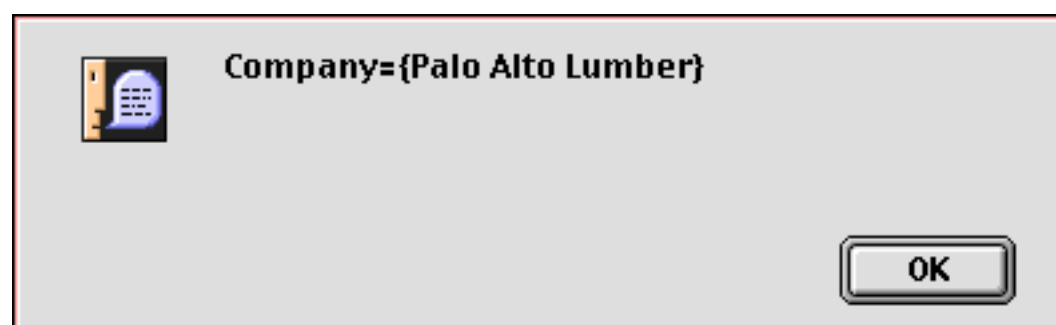
0	1	2	3 enter	4	5	6	7	8	9 tab	10	11	12	13 rtn	14	15
16	17	18	19	20	21	22	23	24	25	26	27 esc	28 left	29 right	30 up	31 down
32	33 !	34 "	35 #	36 \$	37 %	38 &	39 '	40 (41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7	56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G	72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W	88 X	89 Y	90 Z	91 [92 \ backslash	93]	94 ^	95 _
96 `	97 a	98 b	99 c	100 d	101 e	102 f	103 g	104 h	105 i	106 j	107 k	108 l	109 m	110 n	111 o
112 p	113 q	114 r	115 s	116 t	117 u	118 v	119 w	120 x	121 y	122 z	123 {	124 	125 }	126 ~	127 space

As you can see, a regular " quote can be generated with `chr(34)`.

If you are having a difficult time getting your `execute` formula to work properly a good tip is to temporarily replace the execute statement with a `message` statement, like this.

```
local dittoValue,dittoField
dittoField=info("FieldName")
uprecord
if stopped rtn endif
dittoValue=«»
downrecord
message dittoField+="{ "+dittoValue+" }"
```

When the procedure runs it displays the actual subroutine that it was about to execute.



Often at this point the error is obvious and you can easily go back and check it. If the subroutine is too long to fit in the dialog you can copy it into the clipboard instead.

```
local dittoValue,dittoField
dittoField=info("FieldName")
uprecord
if stopped rtn endif
dittoValue=«»
downrecord
clipboard=dittoField+"{"+dittoValue+"}"
```

After the program runs you can paste the generated subroutine into a text editor and carefully examine it.

If the execute was not the last statement in the procedure you will probably want to place a **stop** statement (see “[Stopping the Program](#)” on page 278) after the **message** or **clipboard=** statement.

Execute and Local Variables

The subroutine generated “on-the-fly” by the execute statement is not part of the current procedure but is a separate procedure on its own (although it doesn’t have a name and disappears as soon as it is finished). This means that the local variables in the original procedure become dormant while the “on-the-fly” subroutine is running, and that the “on-the-fly” subroutine cannot use or modify any of those local variables (see “[Variable Accessibility](#)” on page 250). (It can have its own local variables, however.)

To get around this limitation a new statement was added to Panorama in 2004 — the **ExecuteLocal** statement. This statement is identical to the **execute** statement except for the fact that it uses the local variables of the enclosing procedure instead of its own separate local variables. Here is an example:

```
local x
x="Hello"
executelocal {message x}
```

This procedure will display the word “hello”, even though the **x** variable is not defined within the **executelocal** part of the procedure.

Using Execute to Process Arrays

The execute statement can be very handy for working with arrays. For example, suppose you have an comma separated text array (see “[Text Arrays](#)” on page 93) named **Numbers** that contains a series of numbers like this —

```
78,173,9,32,201,12,82,376,249
```

and you’d like to add up the numbers and place the sum in a field named **Total**. With the **execute** statement this can be done with a single line of code!

```
execute "Total="+replace(Numbers,"","+")
```

The **replace()** function (see “[REPLACE\(\)](#)” on page 5665 of the *Panorama Reference*) converts the commas into plus symbols. The generated subroutine is —

```
Total=78+173+9+32+201+12+82+376+249
```

Not only is this code simple, it is very fast. Here is a slightly more complex example that uses a similar technique to calculate the total of an invoice. The invoice looks like this. The big area on the right is a field named **Items**.

Items field

Burritos	A La Carte	Items
Super Deluxe \$4.99	Nachos \$4.99	Taco, Enchilada & Beans \$4.99
Jr. Super Deluxe \$3.99	Cheese & Chips \$3.50	shredded beef
Super \$3.99	Chicken Taco \$1.65	Chile Relleno, Enchilada & Beans \$5.99
Deluxe \$4.99	Shredded Beef Taco \$1.65	Quesadilla \$3.50
Beef \$4.99	Ground Beef Taco \$1.65	Friday Special \$6.00
Beef & Bean \$4.99	Carnitas Taco \$2.00	
Green Chile \$4.99	Carne Azada Taco \$2.60	
Carnitas \$5.50	Chile Relleno \$2.50	
Carne Azada \$5.50	Quesadilla \$3.50	
Bean & Cheese \$2.99	2 Chicken Tacquitos w/quac \$3.50	
Chorizo \$3.99	2 Ground Beef Taquitos w/quac \$3.25	
Chicken \$4.99	Enchilada \$1.75	
Machaca \$3.99	Rice \$1.75	
Relleno \$4.99	Beans \$1.75	
Bean & Egg \$2.99	Potatoes \$1.75	
A La Mexicana \$3.99	Four Tortillas \$1.75	
Tostadas	Corn Tortillas \$1.75	
Carnitas \$4.99	Sour Cream \$1.00	
Carne Azada \$4.99	Guacamole \$2.00	
Chicken \$4.99		
Chunky Beef \$4.99		
Ground Beef \$3.99		
Machaca \$3.99		
Bean \$3.99		
Tortas	Extras	Options
Carnitas \$4.99	potatoes \$0.50	flour
Carne Azada \$4.99	cheese \$0.50	corn
Chicken \$4.99	quacamole \$0.50	shredded beef
Machaca \$3.99	sour cream \$0.50	ground beef
Bean \$3.99	beans \$0.50	chicken
	rice \$0.50	

Clear 20.48
Cancel Item 1.57
22.05

As you can see, if a line in the **Items** field contains a price it is always after a dollar (\$) sign. We can use this fact to quickly calculate the total.

```
fileglobal linecalc
if Items≠" "
  arrayfilter Items,linecalc,¶,array(import(),2,"$")
  linecalc=arraystrip(linecalc,¶)
  linecalc=replace(linecalc,¶,"+")
  execute "Subtotal="+linecalc
else
  Subtotal=0
endif
Tax=(Subtotal*7.75)/100
Total=Subtotal+Tax
```

Lines 3 thru 6 are the guts of this procedure.

Line 3 (`arrayfilter Items,linecalc,¶,array(import(),2,"$")`) uses the **arrayfilter** function (see "**ARRAYFILTER**" on page 5045 of the *Panorama Reference*) to scan the **Items** array and strip out only the prices (after the \$ symbol). The intermediate result in **linecalc** will be something like this.

```
4.99
5.99
3.50
6.00
```

Line 4 (`linecalc=arraystrip(linecalc,¶)`) strips out any extra carriage returns. The result looks now looks something like this.

```
4.99
5.99
3.50
6.00
```

Line 5 (`linecalc=replace(linecalc,¶,"+")`) converts the carriage returns (¶ - see “[Special Characters](#)” on page 57) into plus symbols.

```
4.99+5.99+3.50+6.00
```

Line 6 (`execute "Subtotal="+linecalc`) is just like the previous example and calculates the total.

Do It Yourself Data Merge

Auto-wrap text objects allow you to merge data and formulas into a template to be displayed on the screen or printed in a report (see “[Displaying Data in Auto-Wrap Text](#)” on page 595 of the *Panorama Handbook*). In this section we’ll describe a technique using the `execute` statement that allows a procedure to do the same thing. For example, suppose that you have a template in a variable named `Letter` that contains this text.

```
<Name>
<Address>
<City>, <State> <Zip>
```

```
Dear <Name>,
```

```
Your order (reference <Order>) has been shipped. You should expect it to arrive within the
next three to five days.
```

```
Sincerely,
```

```
Acme Widgets
```

If we can turn this template into a formula we can evaluate it with the `execute` statement. Let’s see how this can be done. The first step is to create two arrays. The first array, `FieldNames`, will contain a list of the fields, like this.

```
<Name>
<Address>
<City>
<State>
<Zip>
<Order>
```

The second array, `FieldFormulas`, will also contain a list of fields, but modified so that they can be included as part of a formula.

```
" }+<Name>+{ "
" }+<Address>+{ "
" }+<City>+{ "
" }+<State>+{ "
" }+<Zip>+{ "
" }+<Order>+{ "
```

Here’s the code that can generate these two arrays (see “[DBINFO\(\)](#)” on page 5150 and “[ARRAYFILTER](#)” on page 5045 of the *Panorama Reference*).

```
local FieldNames,FieldFormulas
FieldNames=dbinfo("fields","")
arrayfilter FieldNames,FieldNames,¶,"<"+import()+">"
arrayfilter FieldNames,FieldFormulas,¶," }+"+import()+"+{ "
```

Now we can use the `replacemultiple()` function (see “[REPLACEMULTIPLE](#)” on page 5667 of the *Panorama Reference*) to transform the template into a formula.

```
local MergeFormula
MergeFormula="{ "+replacemultiple(Letter,FieldNames,FieldFormulas)+" }"
```

After the transformation the template has been turned into a valid Panorama formula.

```
{ }+«Name»+{
}+«Address»+{
}+«City»+{ , }+«State»+{ }+«Zip»+{

Dear }+«Name»+{ ,

Your order (reference }+«Order»+{ ) has been shipped. You should expect it to arrive within
the next three to five days.

Sincerely,

Acme Widgets}
```

Here’s the complete procedure.

```
local FieldNames,FieldFormulas,MergeFormula
fileglobal FinalLetter
FieldNames=dbinfo("fields","")
arrayfilter FieldNames,FieldNames,¶,"«"+import()+"»"
arrayfilter FieldNames,FieldFormulas,¶," "+import()+"{"
MergeFormula="{ "+replacemultiple(Letter,FieldNames,FieldFormulas)+" }"
execute "FinalLetter="+MergeFormula
```

This procedure turns the template into a final letter with all of the data merged in as requested in the template. You could use a procedure like this to generate custom e-mail or as part of a CGI for a web server (in which case the template would contain HTML).

On-The-Fly Subroutine Error Checking

Just as with any other procedure, it’s possible for an on-the-fly subroutine to contain grammar errors. If the subroutine contains a grammar error (for example `a++b`) the procedure will stop and an alert displaying an error message will appear. If you don’t want that to happen you can place an `if error` statement after the `execute` statement. This example procedure executes whatever is in the `PreFlight` field and ignores any grammar error that occurs.

```
execute PreFlight
if error
  nop
endif
```

The program can find out what the problem was with the `info("error")` function (see “[INFO\("ERROR"\)](#)” on page 5373 of the *Panorama Reference*). To find out exactly where the problem occurred check the special global variables `ExecuteErrorStart` and `ExecuteErrorEnd`. These variables contain numbers telling where within the subroutine Panorama thinks the error occurred. If the subroutine was generated with a complex formula it may be difficult to relate these numbers back to the formula that generated the subroutine.

The `if error` statement after the `execute` only catches grammar errors, not run-time errors. If you want to catch run-time errors (for example `field or variable does not exist` or `numeric when text expected`) you must build if error statements into the generated subroutine itself (see “[Error Handling with if error](#)” on page 258).

Building Parameters on the Fly (Parameters in a Variable)

Many statements accept one or more formulas as parameters. These formulas are usually fixed at the time the procedure is written. For example the statement below has four parameters:

```
arrayfilter a,b,¶,str(seq())+": "+import()
```

Most statements also allow you to store the text of the formula itself in a variable. To do this, use `@variable` or `@«variable»` instead of the formula itself. For example, the previous example could be rewritten using this technique:

```
local myFormula
myFormula={str(seq())+": "+import()}
arrayfilter a,b,¶,@myFormula
```

In this example there is no advantage to this technique, but the power is that now you can change the formula "on the fly". Of course you could also do this with the `Execute` statement, like this:

```
local myFormula
myFormula={str(seq())+": "+import()}
execute {arrayfilter a,b,¶,}+myFormula
```

Using the `@` technique has two benefits over using the `Execute` statement. First of all, it can be quite a bit faster. Secondly, it makes it much easier to work with local variables.

Warning: When using this technique, take extra care in checking your formulas. Panorama often cannot accurately report the source of a formula error when this technique is used.

Catching Program Errors (Especially for Web and other Server Applications)

The `if error` statement (see "[Error Handling with if error](#)" on page 258) gives the programmer complete control of what happens when an error occurs at a specific point in the program. However it requires the programmer to explicitly handle every error that may occur. The `OnError` statement can be used to catch all errors that are not trapped by `if error` statements. This has two benefits when Panorama is used as part of a web server. First it allows the programmer to easily eliminate all error alert dialogs. This is very important for server applications because an alert dialog requires human intervention to get the server going again. Secondly, it makes it easy to build a log of errors.

The `OnError` statement has one parameter: a text string that contains one or more Panorama statements to be executed when an error occurs. Notice that this is not the name of a procedure, but the actual statements themselves (as a string of text). This is similar to the `execute` statement (see "[Building Subroutines On The Fly \(The Execute Statement\)](#)" on page 280). Once an error has occurred these statements will run. Within these statements you can use the `info("error")` function to find out what the error was, if necessary.

The effect of the `OnError` statement ends when the main procedure stops running. In other words, `OnError` isn't a permanent error handler — you must specify it for each procedure you wish to have error trapping. If you plan to use `OnError`, it is probably best to put it in the first line of any procedure that needs error trapping. If you are going to use the same statements with `OnError` in several different procedures, you may want to set up the statements in a variable in your `.Initialize` procedure, then use that variable as the parameter to `OnError`.

It's important to consider the possible environment that may exist when an error is created. Depending on the flow of your main procedure, Panorama may not be in the same window or even in the same database. Your `OnError` program should generally not make any assumptions about what windows or databases will be active or available when the error occurs.

Here is an example of how **OnError** could be used in a CGI (web server) application. In this example if there is an error Panorama will return an error message to the web server and also log the error along with the date and time.

```
global cgiResult,errorLog
errorLog=errorLog          /* make sure errorLog exists */
if error
    errorLog=""            /* initialize errorLog */
endif
onerror {cgiResult="Panorama Error: "+info("error") }+
        {errorLog=sandwich(" ",errorLog,¶)+}+
        {datepattern(today(),"DD/MM/YYYY ")+}+
        {timepattern(now(),"hh:mm:ss")+}+
        {info("error")}

/* error logging is set up, now we can continue with our tasks */
...
... rest of this procedure
...
```

Custom Statements

Panorama comes with hundreds of ready to use built-in statements (see “[Programming Reference Wizard](#)” on page 237), but you aren’t limited to these built-in statements. If you don’t find the statement you need you can build your own! If you are planning on using a particular sequence of steps frequently then it might pay to create a custom statement that you can use in any database.

As you might have guessed from the phrase “sequence of steps” in the previous paragraph, custom statements are very similar to subroutines. In fact, custom statements actually are Panorama subroutines, written using Panorama’s standard procedure editor and the Panorama programming language. The only difference is that these subroutines are placed in a special location, where Panorama automatically loads them so. As it loads the database containing these special subroutines, Panorama also adds them to the programming language so that they are always available to procedures in any database.

To illustrate this, consider the subroutine shown below, which will make a new folder. This procedure is called **MAKENEWFOLDER** and is contained in the **_DiskLib** database.



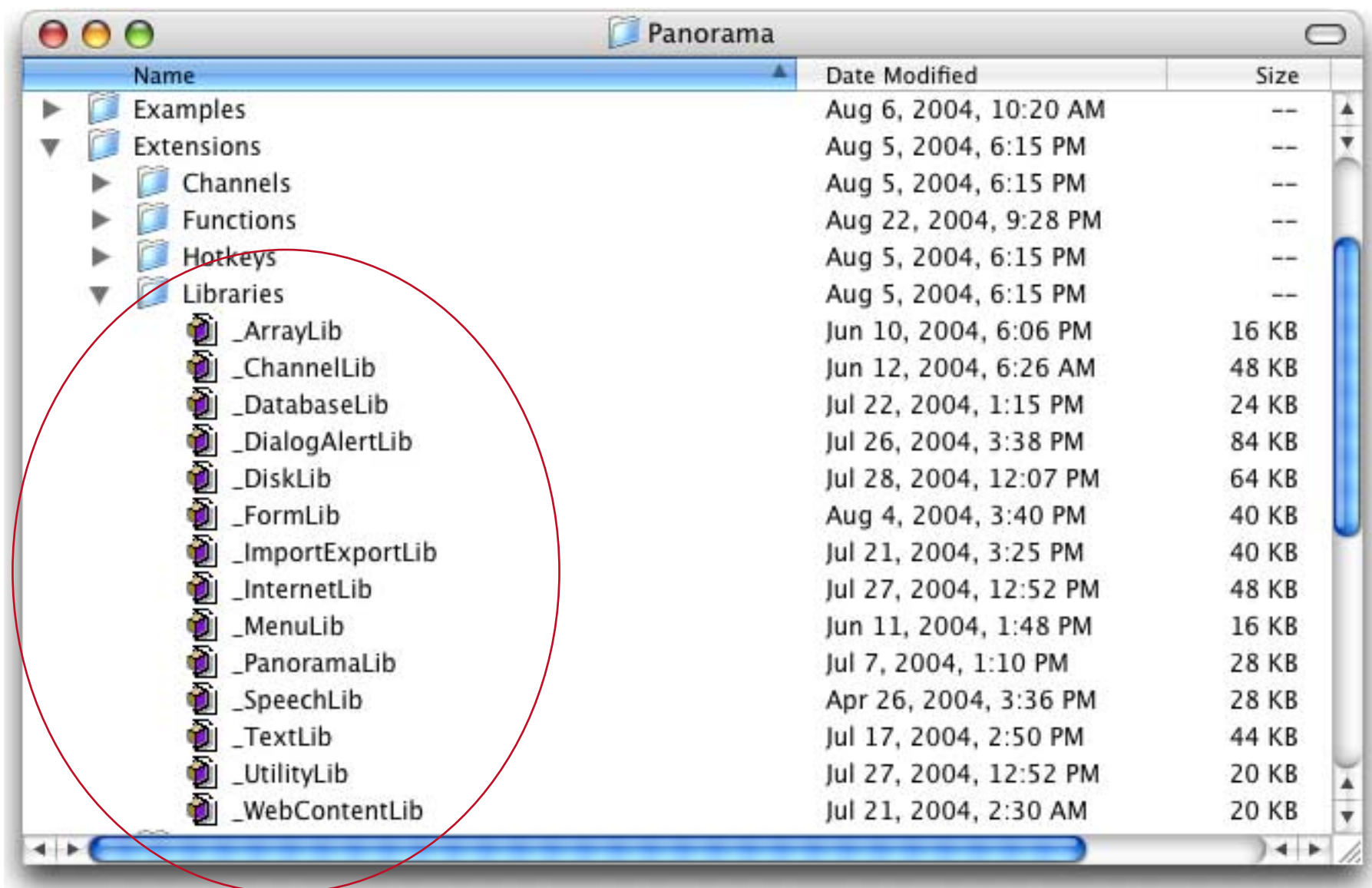
Even without the ability to create custom statements you can still call this subroutine from another database using the line shown below. (see “[Calling a Subroutine in Another Database](#)” on page 269).

```
farcall "_DiskLib", "MAKENEWFOLDER", "My Drive:My Documents:MyImages:Zack:"
```

This technique has some drawbacks — you have to remember that this subroutine is in the `_DiskLib` database, you have to make sure that the `_DiskLib` database has actually been opened, and you have to use the exact capitalization `MAKENEWFOLDER` because `MakeNewFolder` or `makenewfolder` won't work. Converting this procedure into a custom statement fixes these drawbacks. Here is the same subroutine used as a custom statement:

```
MakeNewFolder "My Drive:My Documents:MyImages:Zack:"
```

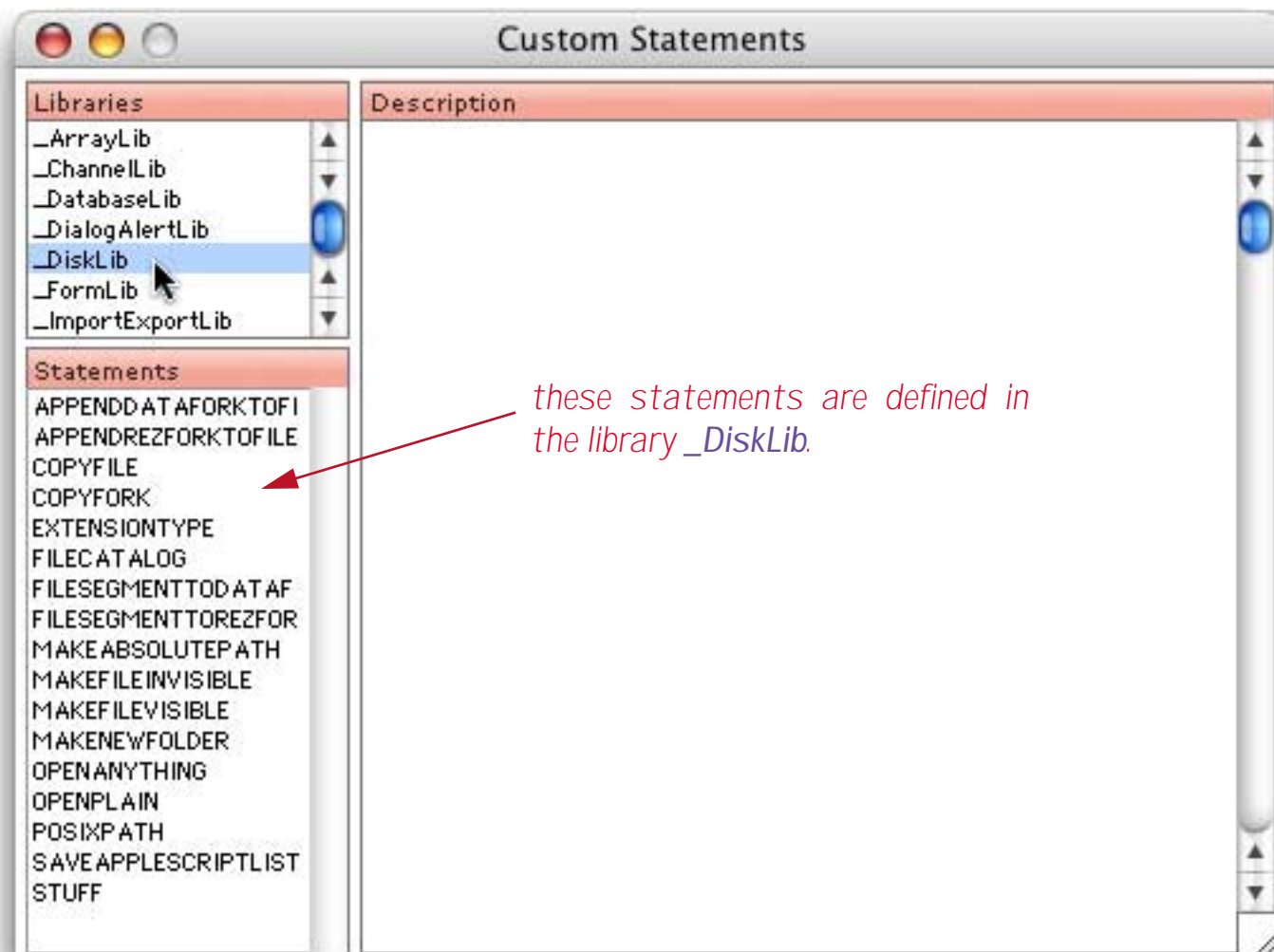
As illustrated in this example, the procedures used in custom statements are stored in one or more Panorama databases. These databases are called **procedure libraries**. These library databases are normally kept in the `Extensions:Libraries` folder within the Panorama folder. (Note: All of the databases in the folder shown below begin with `_` and end with `Lib`, but that is not necessary. You can use any name you like as long as the database is placed within this folder.)



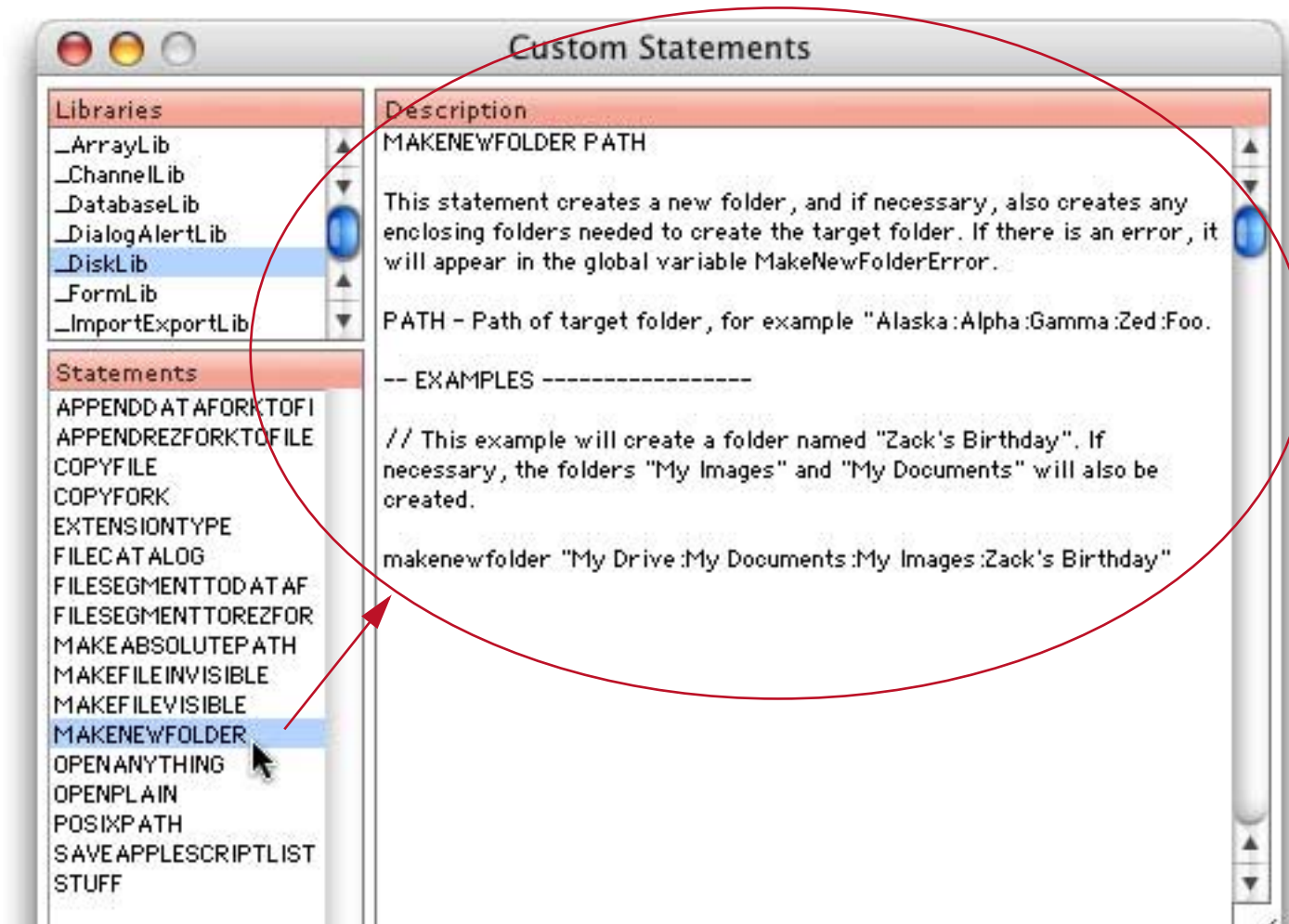
If a database is placed in this folder Panorama will automatically open it when Panorama launches (the database is opened secretly, with no windows). Panorama will then scan the database and register each procedure in the database as a custom statement. (Actually, only procedures with names that are all upper case letters will be registered, for example `ZIP` or `ZAP` but not `Zip` or `ZAP23`.)

The Custom Statements Wizard

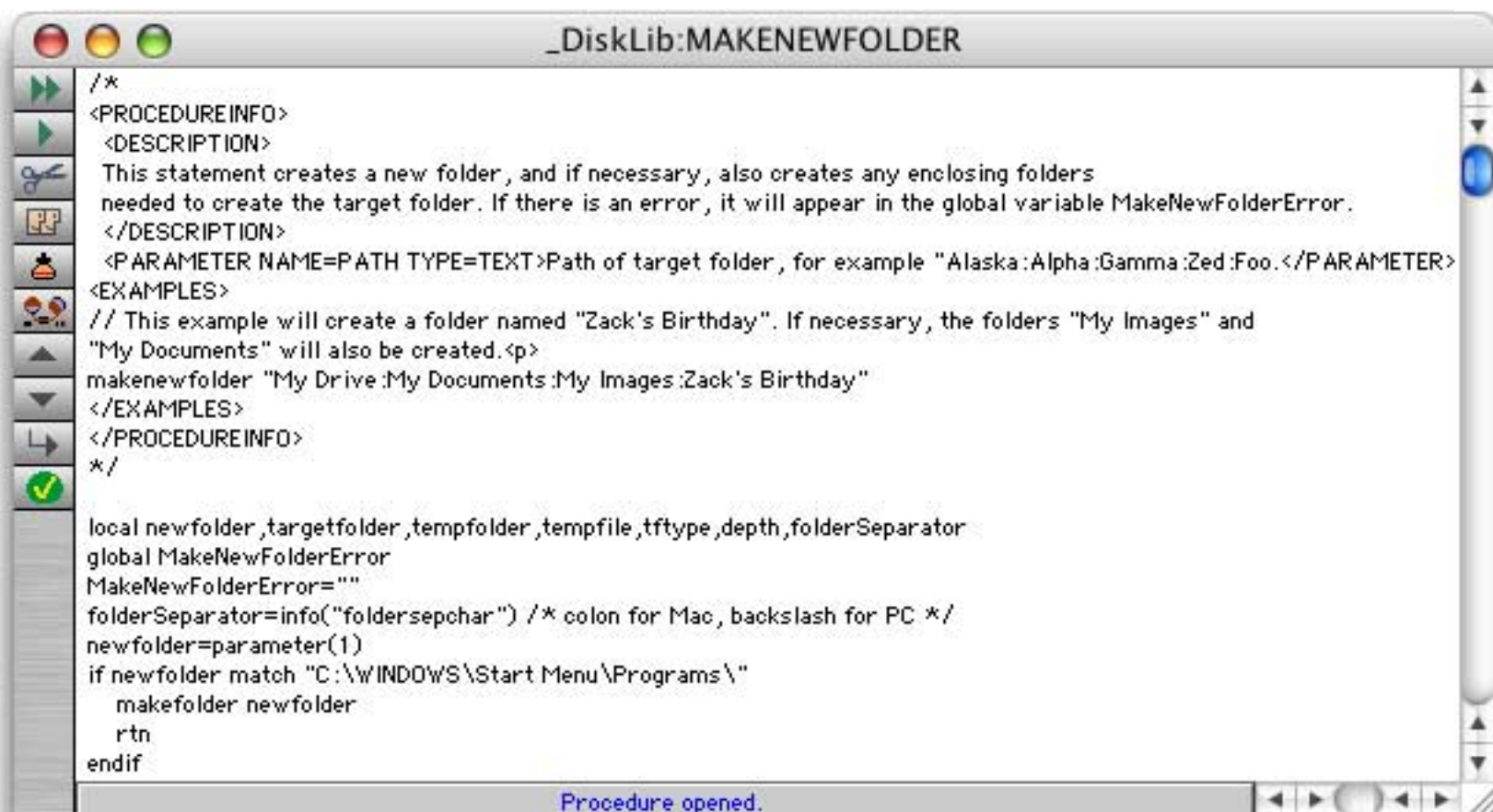
To learn about the custom library statements that are available for your use you can use the **Custom Statement** wizard. When you open this wizard it displays a list of the libraries that are currently available (on the top left). When you click on a library in this list, the wizard displays a list of statements defined in that library (on the bottom left).



You can click on a specific statement to see a description of that statement and its parameters.



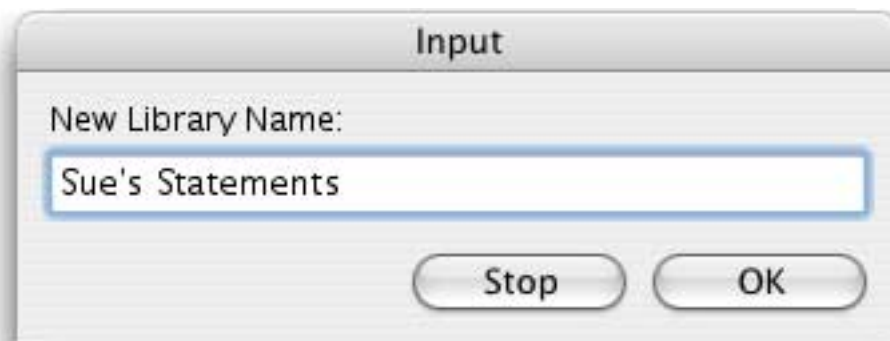
If you want to see (or even edit) the actual code of this statement, double click on the statement name (in the list of statements). The actual code for this statement will appear in a separate window. (Note: If the source code for a custom statement has been opened recently you can also re-open it directly from the **Recent** menu.)



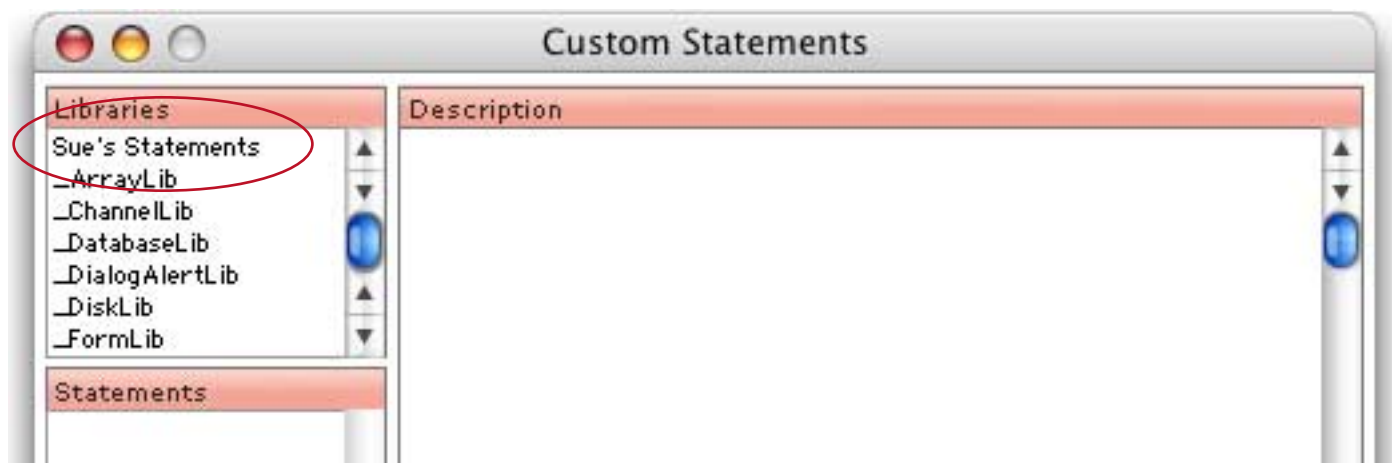
If you make any changes to a statement, please do so carefully (especially for statements that are supplied with Panorama). When you close this window the changes are automatically saved.

Creating Your Own Custom Statement Library

Panorama comes with about a dozen pre-built custom libraries that you can begin using right away, but you can also use the **Custom Statements** wizard to create your own. To do so, choose **New Library** from the Library menu. (You may have noticed that all of the libraries that come with Panorama have names that begin with `_` and end with `Lib`, but that is not necessary.)



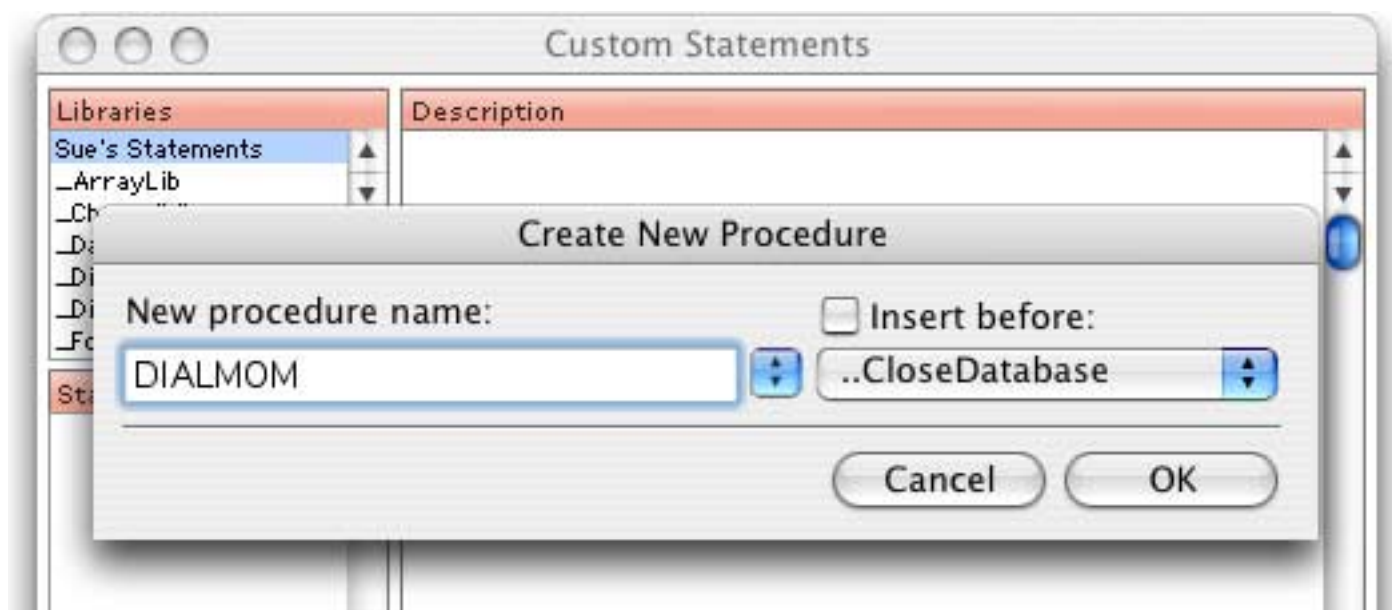
Once you've entered the name, press **OK**. Your new library will appear in the list of libraries in the upper left hand corner.



If you click on your new library you'll see that it doesn't contain any statements yet.

Creating a New Custom Statement

To create a new custom statement, first make sure that you have clicked on the name of the library you want the statement to be created in. ((Note: We do not recommend that you add new statements to the libraries that come with Panorama. If you do so, your new statements will be destroyed the next time you update Panorama. Instead, make sure to add your new statements to your own libraries.) Then choose **New Statement** from the Statement menu, and type in a name for your new statement.



The name must contain only the letters **A** through **Z** (uppercase only). You cannot include spaces, numbers, lower case letters, or any other kind of character except for upper case letters. (Note: The **New Statement** command won't stop you from entering these characters, but if you do, you cannot use this procedure as a custom statement.) Here are some examples of procedure names that may be used for custom statements.

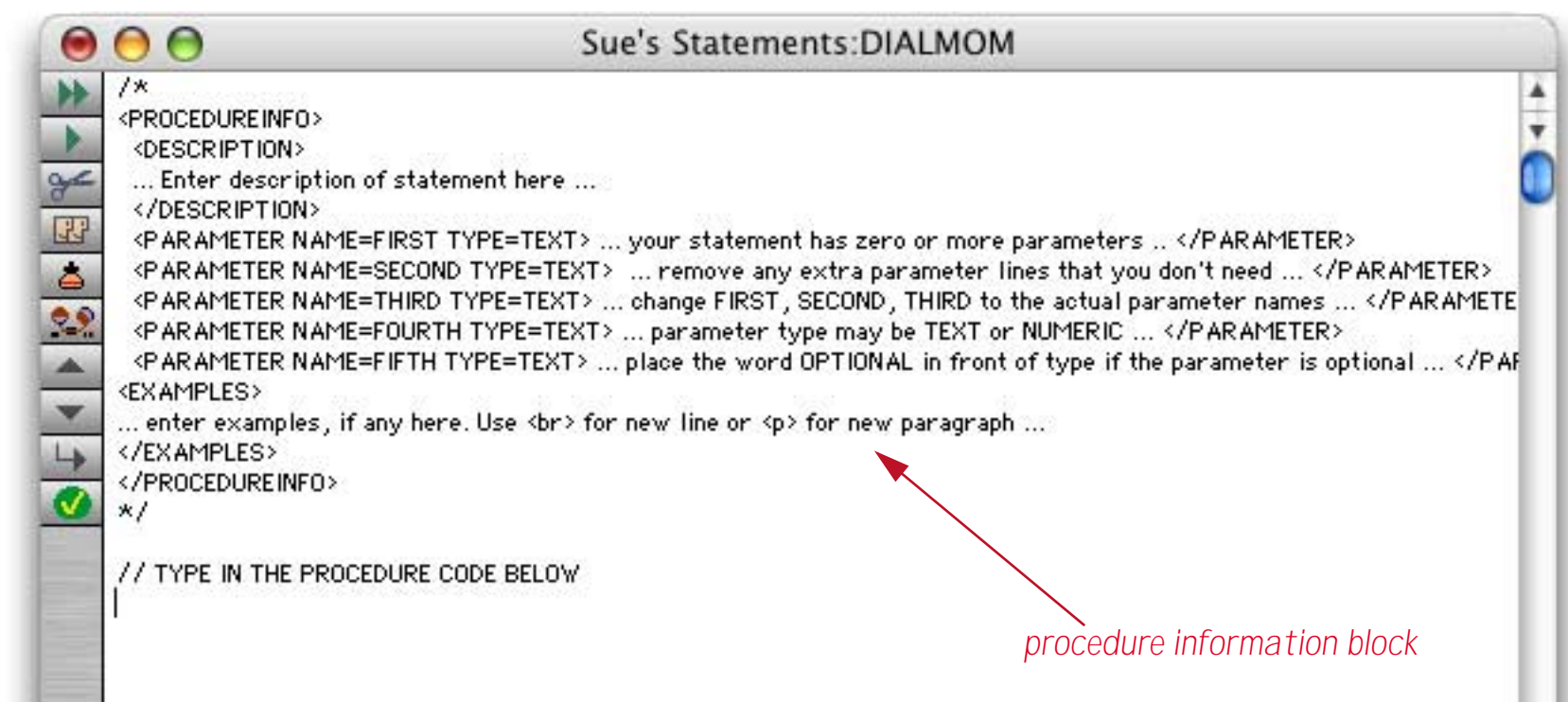
```
PARSENAME
SPLITINPUT
WRITELOG
```

Here are some names that *may not* be used for custom statements. (If your library database contains procedures with these names those procedures will be ignored when the library is initialized.)

```
ParseName
SPLIT INPUT
CARD#
```

To avoid conflicts with future versions of Panorama it's best to choose names that are unlikely to be used by Panorama itself. For example, **MYWRITELOG** or **ACCOUNTINGLOG** might be a safer choice than **WRITELOG**. If a future version of Panorama includes a statement with the same name as your statement then your statement will stop working (you'll have to rename your statement and locate and change every place that you have used it).

OK, back to the action. When you press the **OK** button, the wizard will create a brand new procedure with the specified name.



Although it's not required, it's a good idea to include a **procedure information block** in each procedure that will become a statement. In fact this is such a good idea that when you create a new statement a "typical" procedure information block is created for you, as shown above. However, for this simple example we won't use the information block, so erase it and type in the actual code of the procedure.



This simple statement simply dials Mom's phone number. However, the statement isn't quite ready to use. The final step is to "register" the statement with Panorama. To do this, click back to **Custom Statements** wizard and choose **Register New/Modified Statements** from the Statements menu. The new statement will appear in the list of statements, and is now ready to use. (Notice that in the description area it says **No additional information is available**. This is because there is no **Procedure Information Block** for this statement.)



To use this new statement, simply type it into any procedure in any database. (Notice that like any other statement, the **dialmom** statement is not sensitive, you can use **DIALMOM**, **dialmom**, **DialMom**, **diALmOM** or any other combination of upper and lower case letters.)



That's all there is to it. Later, if you quit and then re-launch Panorama, this statement will automatically be registered as part of Panorama's initialization process. From now on the **dialmom** statement is always available whenever you need it. (Of course you can always modify and/or delete it later if you wish, but be careful, as this will break any procedures that use your new statement.)

Setting Up a Procedure Information Block

Although it's not required, it's a good idea to include a **procedure information block** in each procedure that will become a statement. This block contains descriptive information about the statement and its parameters. This serves two purposes:

- 1) It allows the statement to be "self-documenting" through the Custom Statements wizard.
- 2) It allows Panorama to perform some error checking on the parameters of your statement, making debugging easier. For example, when this statement is used in a procedure Panorama can check to make sure that the proper number of parameters is supplied.

The procedure information block should be included in a comment within your procedure (usually at the top) and looks something like this.

```

/*
<PROCEDUREINFO>
<DESCRIPTION>
--- put description of procedure here ---
</DESCRIPTION>
<PARAMETER NAME=name TYPE=TEXT>--- put description of parameter here</PARAMETER>
<PARAMETER NAME=name TYPE=TEXT>--- put description of parameter here</PARAMETER>
<PARAMETER NAME=name TYPE=TEXT>--- put description of parameter here</PARAMETER>
<EXAMPLES>
--- put examples (if any) here
</EXAMPLES>
</PROCEDUREINFO>
*/

```

When you create a new statement a "typical" procedure information block is created for you. You'll need to modify this block to reflect the actual structure of your new statement. The most important part is to set up the parameter list to have the correct number of parameters.

Each parameter has a name, type and description.

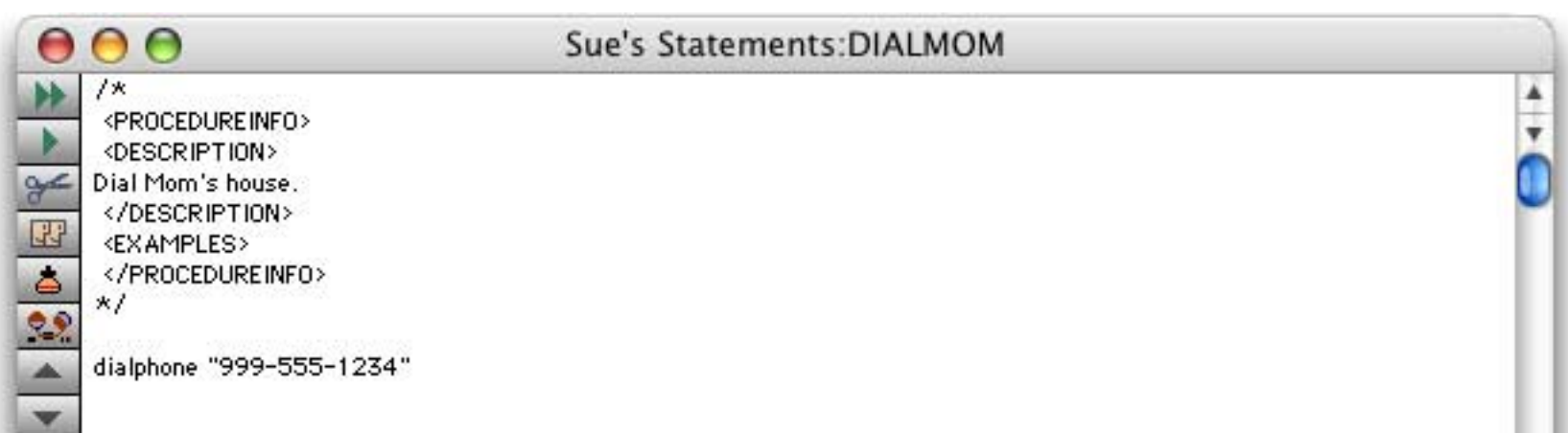
```
<PARAMETER NAME=name TYPE=TEXT>--- put description of parameter here</PARAMETER>
```

The current version of Panorama only supports one type: **TEXT**. The name is used only for documentation purposes.

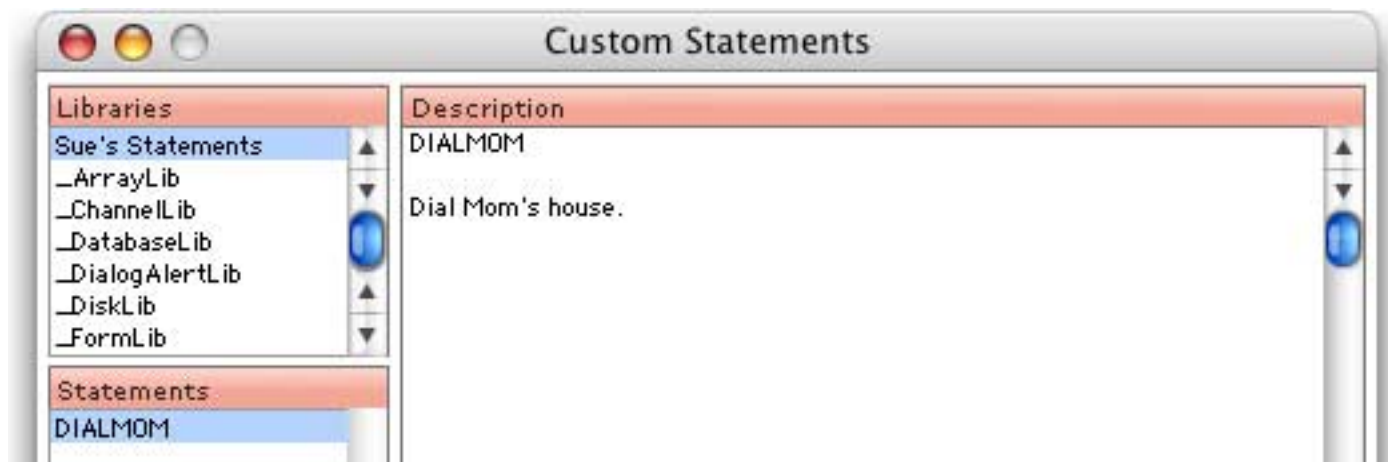
In the description area any extra whitespace will be ignored. In other words, when the description is displayed carriage returns will be turned into spaces and anywhere there are two or more spaces in a row they will be replaced by a single space (just like HTML text).

The examples section is optional. If included, whitespace is handled the same as the description area. However, you can also include `<p>` and `
` tags which work just like HTML. (However, `<P>` and `
` are not supported, the tags must be lower case.)

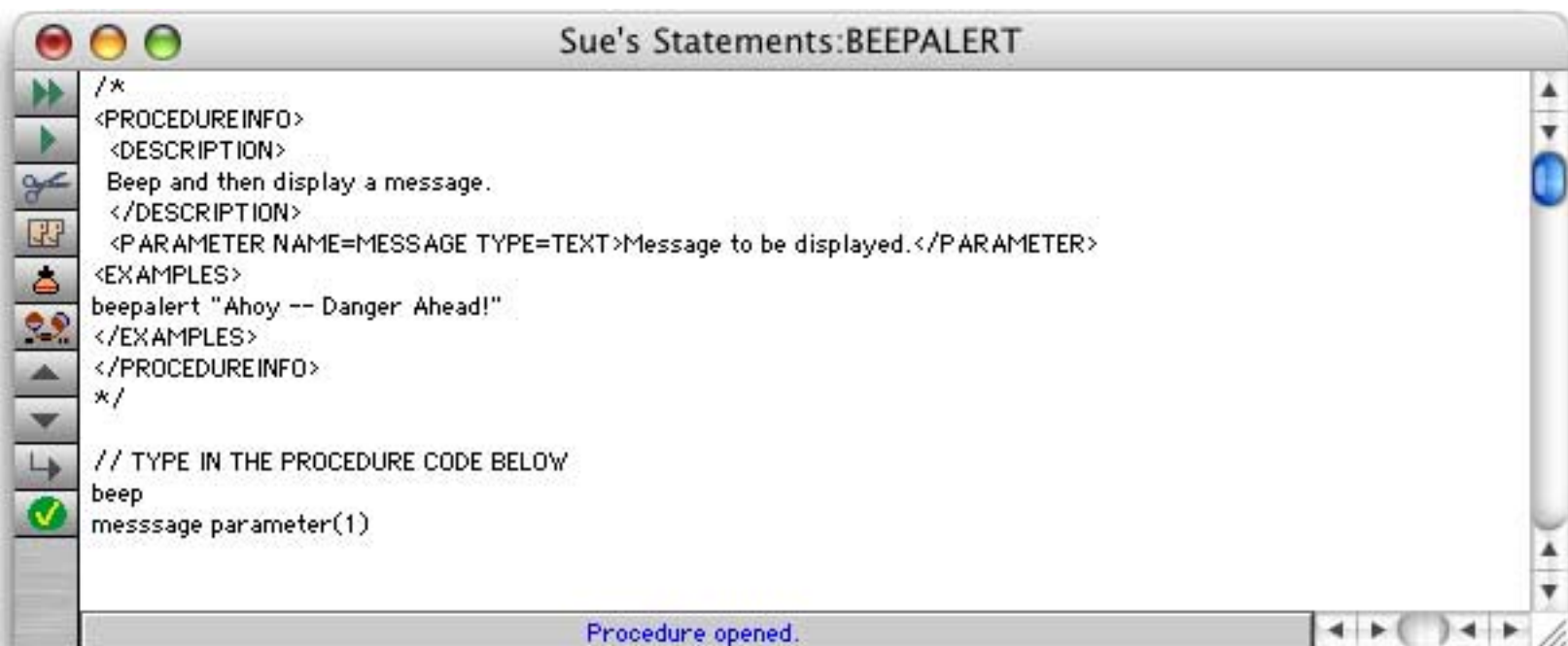
In the previous section a simple **DIALMOM** statement was created. This statement has no parameters at all, so the procedure information block would look like this:



Whenever you change the procedure information block you must go back to the Custom Statements wizard and use the **Register New/Modified Statements** command from the Statements menu. This is especially true if you change the number of parameters. This command tells Panorama to update its internal description of this statement. After using this command the wizard now shows the description from the information block.



Here is a slightly more advanced custom statement. This statement has one parameter, a message to be displayed.



After the **Register New/Modified Statements** command is used, this statement will appear in the Custom Statements wizard like this.



If your procedure has additional parameters you need to document each one in the procedure information block.

Processing Parameters

Within a custom statement parameters are usually handled with the `parameter()` function and `setparameter` statement, just like a normal subroutine (see “[Passing Values to a Subroutine \(Parameters\)](#)” on page 263 and “[Passing Values Back From a Procedure](#)” on page 264). For example, here is a custom statement that uses two parameters.

```

/*
<PROCEDUREINFO>
<DESCRIPTION>
This statement strips blank elements from an array.
</DESCRIPTION>
<PARAMETER NAME=ARRAY TYPE=TEXT>Array to be stripped.</PARAMETER>
<PARAMETER NAME=SEPARATOR TYPE=TEXT>Separator character.</PARAMETER>
<EXAMPLES>
// This example strips empty elements from an array.<p>
arraystrip elements,[]
</EXAMPLES>
</PROCEDUREINFO>
*/

if length(parameter(2))<>1
  rtnerror "ArrayStrip: Separator must be a single character."
endif
setparameter 1,arraystrip(parameter(1),parameter(2))

```

get original array value
get array separator character
set new array value

Procedure opened.

The custom statement libraries that come with Panorama are full of ideas and techniques that you can use in your own custom statements.

Optional Parameters

In some cases a parameter may be optional. If the parameter is not supplied then the statement will assume a default value. To make a parameter optional you need to adjust the procedure information block and the procedure itself. For example, the custom statement below has one parameter (the date), but it is optional.

```

/*
<PROCEDUREINFO>
<DESCRIPTION>
Add a new transaction to the current database.
</DESCRIPTION>
<PARAMETER NAME=DATE OPTIONAL TYPE=TEXT>Date for this new transaction (leave off for today).</PARAMETER>
<EXAMPLES>
newtransaction "4/12/04"
</EXAMPLES>
</PROCEDUREINFO>
*/

// TYPE IN THE PROCEDURE CODE BELOW
addrecord
Date=parameter(1)
if error
  Date=today()
endif

```

Procedure OK.

To make this parameter optional, the word **OPTIONAL** may be placed anywhere within the `<parameter...>` tag, as shown above. Since the parameter is now optional, the `parameter()` function might not work. The procedure checks for this with the `if error` statement (see “[Error Handling with if error](#)” on page 258). If the parameter is missing, this particular statement substitutes today’s date.

Because the parameter is optional, the `newtransaction` statement can be used with or without a date parameter. If there is no parameter, the statement assumes today’s date.

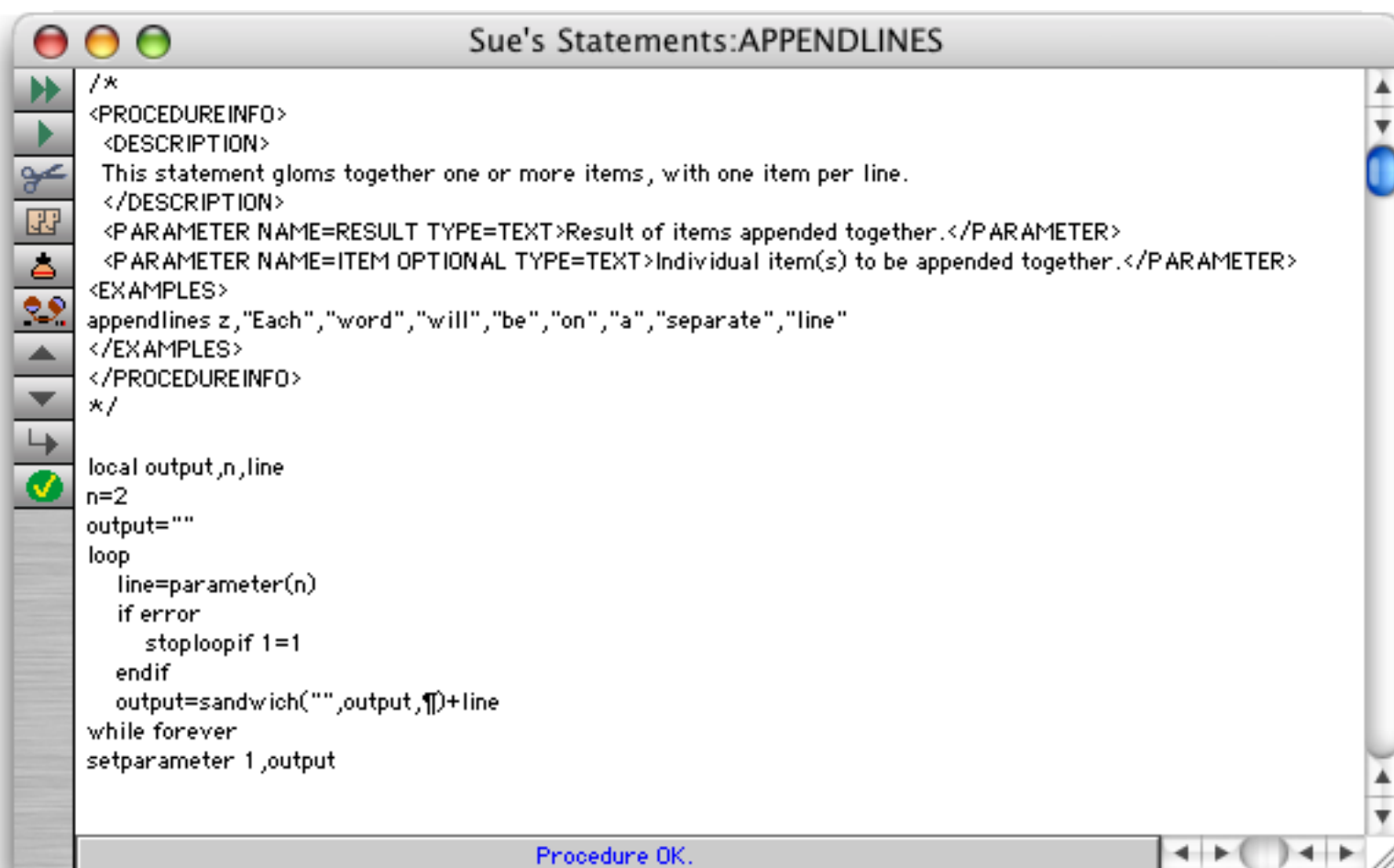
```
newtransaction
```

If there is a parameter, the statement will use that date instead.

```
newtransaction "8/27/03"
```

Repeating Parameters

Sometimes you may not know in advance how many parameters are needed. If placed at the end of the parameter list, an optional parameter can actually repeat over and over again. The second parameter of this custom statement is optional, so it can repeat over and over again.



Notice that the loop loads increasing parameters (2, 3, 4) until it finds a missing parameter, at which point it stops.

Raw Parameters

The `parameter()` function allows a procedure to determine the value of a parameter. Sometimes, however, a procedure needs to access the text of the parameter itself, rather than the value. This information is available in the `<<_RawParameters>>` local variable. This variable is a tab separated array with the raw text of all of the parameters passed to the subroutine.

If a statement needs to access the raw text of a parameter you'll usually need to add the keyword **RAW** to the procedure information block for that parameter, like this:

```
<PARAMETER NAME=ASSIGNMENT RAW TYPE=TEXT>Assignment statement.</PARAMETER>
```

The **RAW** option tells Panorama not to display an error message if it can't compute a value for the parameter (for example if a variable hasn't been defined yet). However, each parameter must still be a syntactically correct formula. In other words, it's ok to have a parameter of **x** even if **x** hasn't been created yet, but **x+** can never be a legal parameter because that is a syntax error.

To see some good examples of how raw parameters can be used check out the **MAKEGLOBAL** and **ARRAYSUBSET** custom statements in the **_UtilityLib** library.

Debugging a Custom Statement

A custom statement can be debugged like any other procedure. You can put a **debug** statement in the code and then single step through it (see "[Debugging a Procedure](#)" on page 312). As long as the procedure editor window is open you'll be able to step through the code. You can even step through the custom statements that are supplied with Panorama.

Accessing Forms & Procedures in the Library Database

When you write a custom statement keep in mind that as the procedure runs the current database is normally the database that has the topmost window... not the library database. If you need to refer to an item in the current database you can use the **info("proceduredatabase")** function. For example, suppose you want to open a form called "my form" in the library database. Here's an example of how this could be done:

```
window info("proceduredatabase")+":SECRET"
openform "my form"
```

Here's how to access a fileglobal or permanent variable in the library database:

```
grabfilevariable(info("proceduredatabase"),"variableName")
```

Here's how to call another procedure in the library database:

```
farcall info("proceduredatabase"),theprocedurename,parameters,if,any
```

Of course if the other procedure has a name that is all capital letters it will be a statement and can be called that way as well.

Advanced Topic: Using Libraries In Other Folders

Panorama automatically loads the libraries in the Extensions:Libraries folder when Panorama opens. However, it is possible to have libraries in other locations. For example, you might want to do this for a commercial product so that you can locate the library in your own folder instead of having to install it into the Panorama folder. In this case, however, you must manually load the library. Naturally you do this with a custom statement - the **LoadLibrary** statement. This statement has two parameters, the path to the folder containing the library and the name of the library itself. Here's how you could load the library database My Library which is in the same folder as the current database.

```
loadlibrary folderpath(dbinfo("folder","")), "My Library"
```

(By the way, there's no harm in loading a library more than once, except for the small delay. If you want, you can check to see if the library is already loaded with the **info("procedurelibraries")** function.) This function returns a carriage return separated list of all of the libraries that are currently active.

Program Formatting

The way a program is formatted can make a big difference in how understandable it is. Panorama is very flexible in letting you format a program; you can have multiple statements on a single line, or split a single statement over multiple lines. Statements can be flush on the left or indented; it's all up to you.

For example, here's a sample procedure from earlier in this chapter with one line per statement.

```
select Debit>0
field Category
groupup
field Debit
total
outlinelevel 1
```

Here's the same procedure squished onto a single line. Panorama will understand this just fine, although you and I might have a more difficult time.

```
select Debit>0 field Category groupup field Debit total outlinelevel 1
```

You can even split individual statements across multiple lines, as long as you don't split a single word or constant in the middle. Here's another version of this same program.

```
select
Debit>0
field
Category
groupup
field
Debit
total
outlinelevel
1
```

Anyplace you can have a single blank or carriage return you can have more than one. Here's one final example.

```
select      Debit>0
field      Category
groupup
field      Debit
total
outlinelevel 1
```

In general, we recommend using one statement per line for readability. We also recommend indenting the statements between **if** and **endif**, **case** and **endcase**, and between **loop** and **until** or **while** (as seen in most of the examples throughout this manual). If you have multiple levels of nested **if** statements, each level should be indented further. This makes it easy to see which statements are associated with each **if/endif** pair.

Here's a program example with no indenting:

```
local CardLength
if PaymentMethod="Credit Card"
CardLength=length(CardNumber)
if CardLength<13 or CardLength>16
message "Sorry, invalid credit card number."
endif endif
```

Now the same procedure with the recommended indenting:

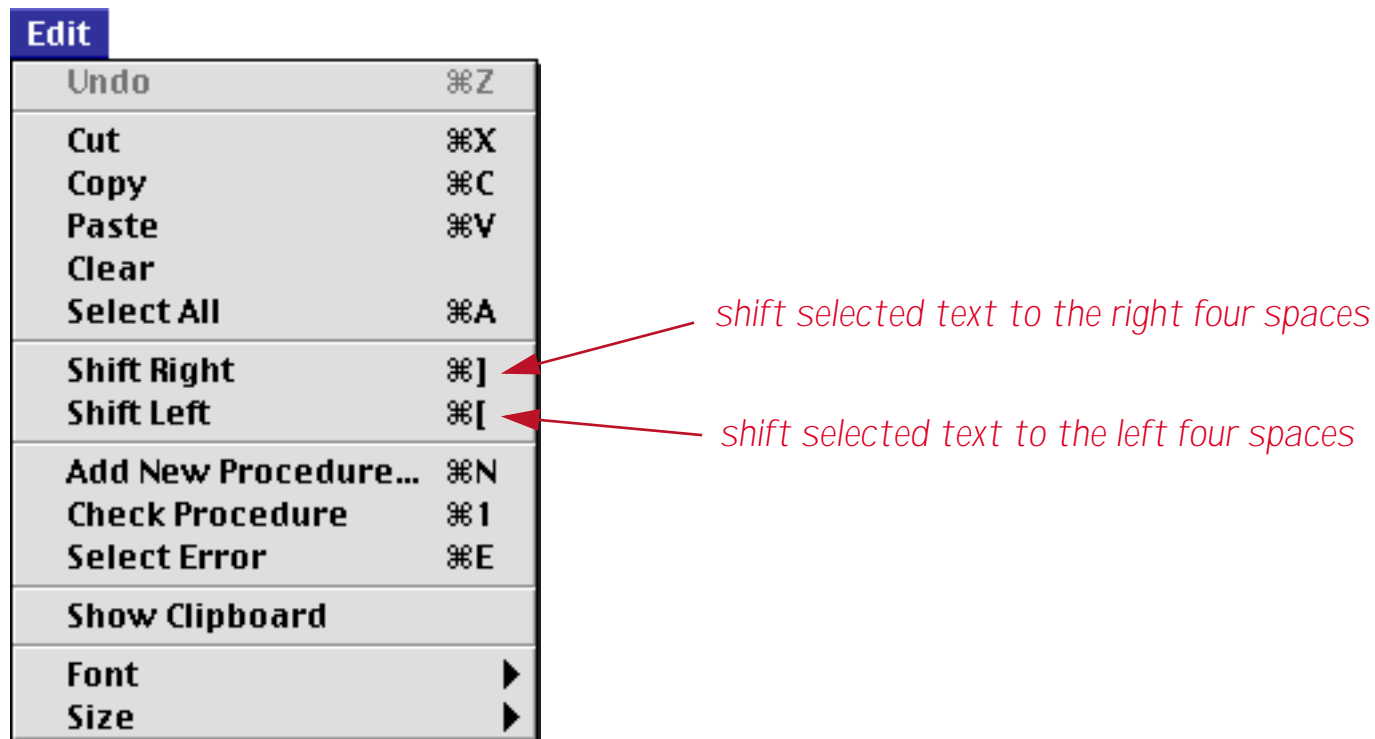
```

local CardLength
if PaymentMethod="Credit Card"
  CardLength=length(CardNumber)
  if CardLength<13 or CardLength>16
    message "Sorry, invalid credit card number."
  endif
endif
endif

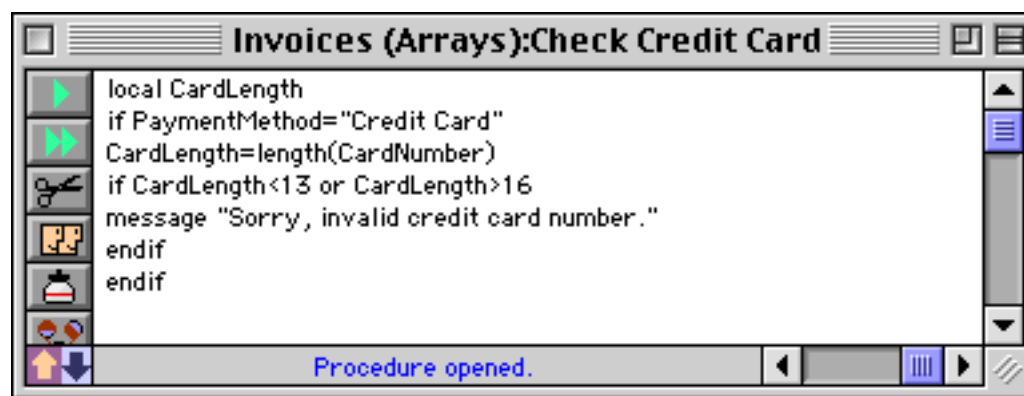
```

A lot easier to understand, isn't it?

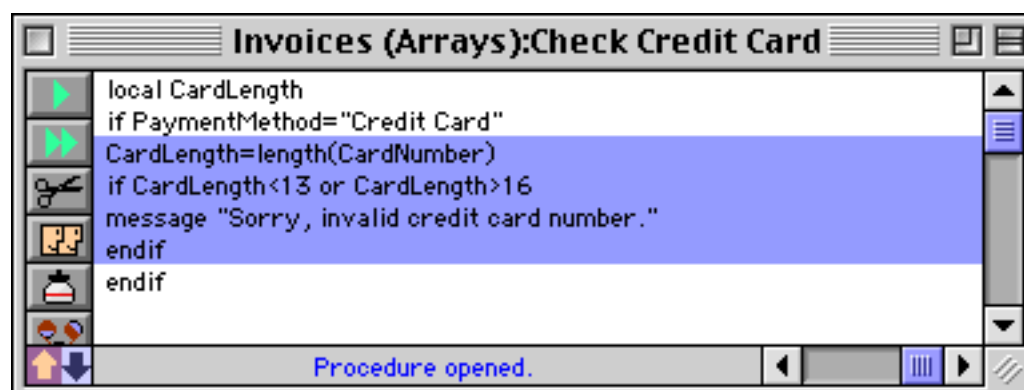
The **Edit** menu has two commands that can help you shift a section of text to the left or right.



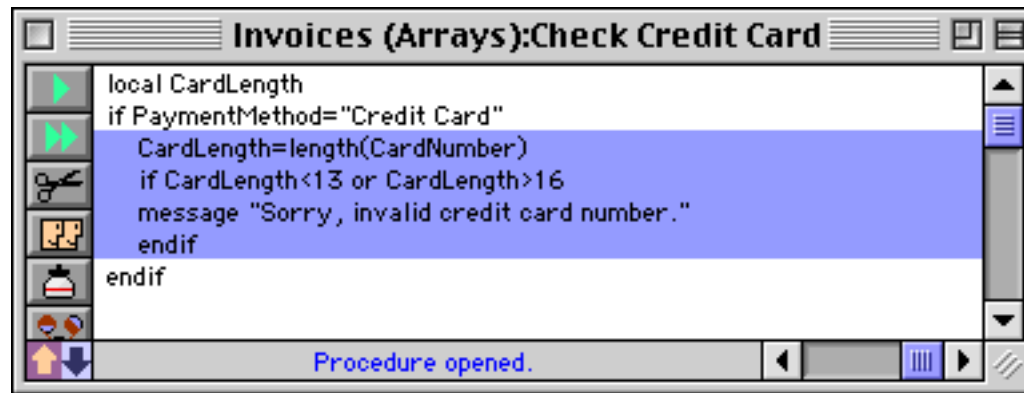
For example, suppose you start with this procedure, which is not indented at all (it's the same procedure listed earlier in this section).



To indent the text, start by selecting the text between the **if** and **endif** statements.



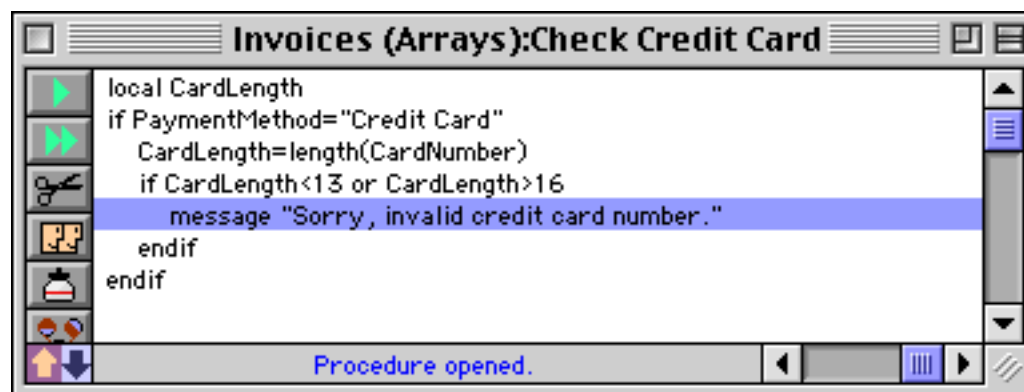
Now use the **Shift Right** command to indent the selected text.



```
local CardLength
if PaymentMethod="Credit Card"
  CardLength=length(CardNumber)
  if CardLength<13 or CardLength>16
    message "Sorry, invalid credit card number."
  endif
endif
```

Procedure opened.

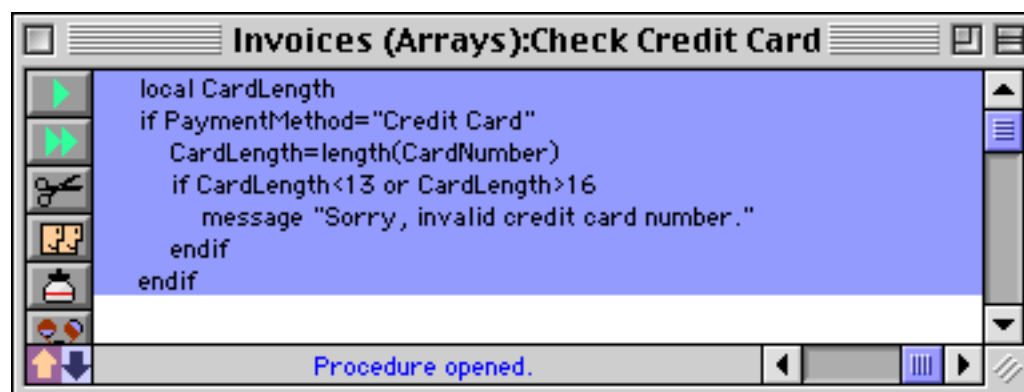
Repeat as necessary to indent any other text.



```
local CardLength
if PaymentMethod="Credit Card"
  CardLength=length(CardNumber)
  if CardLength<13 or CardLength>16
    message "Sorry, invalid credit card number."
  endif
endif
```

Procedure opened.

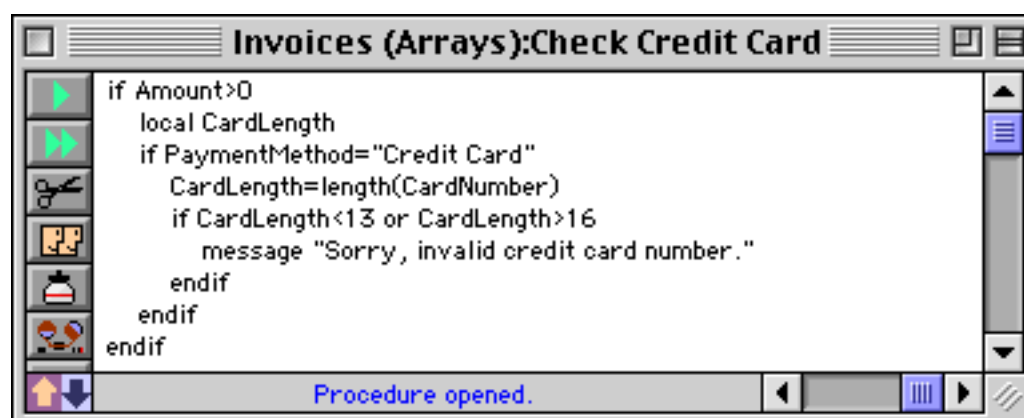
You can even use these commands to change the indentation of the entire procedure.



```
local CardLength
if PaymentMethod="Credit Card"
  CardLength=length(CardNumber)
  if CardLength<13 or CardLength>16
    message "Sorry, invalid credit card number."
  endif
endif
```

Procedure opened.

Now we can add another if statement around the entire procedure.



```
if Amount>0
  local CardLength
  if PaymentMethod="Credit Card"
    CardLength=length(CardNumber)
    if CardLength<13 or CardLength>16
      message "Sorry, invalid credit card number."
    endif
  endif
endif
```

Procedure opened.

Consistent indentation can go a long way towards making your programs more readable and bug free.

Notes To Yourself

A **comment** is a note inside the program. Comments are very useful for documenting how a procedure works, what the variables are for, what the procedure parameters are, etc. Comments are totally optional, but you should use them to record anything you think you might forget about the operation of a procedure.

Panorama has three different comment styles: `/* ... */`, `//`, and `;`.

`/* ... */` comments begin with `/*` and end with `*/`. The advantage of this type of comment is that a single comment may be many lines long. You can also use this type of comment within a formula.

`//` comments begin with `//` and continue to the end of the line.

`;` comments begin with a semicolon and continue to the end of the line.

A comment can appear almost anywhere in a procedure. The only restriction on comments is that they cannot be inside the middle of a statement or formula; they must be between statements.

This example shows a procedure with lots of comments. If anyone comes back and takes a look at this procedure next year, they will have no problem telling what the procedure does and how it does it. To emphasize the comments they are shown in purple below. However, when actually editing a procedure the comments are black just like everything else.

```

/* Procedure: .Delay

This procedure delays for a fairly precise time.
The procedure has one parameter, the number of seconds to delay.

Example:
  call .Delay,12; delay for 12 seconds
*/
local startTime
startTime=now()// record the time we started
loop
  nop ; short delay
until now(>startTime+parameter(1)

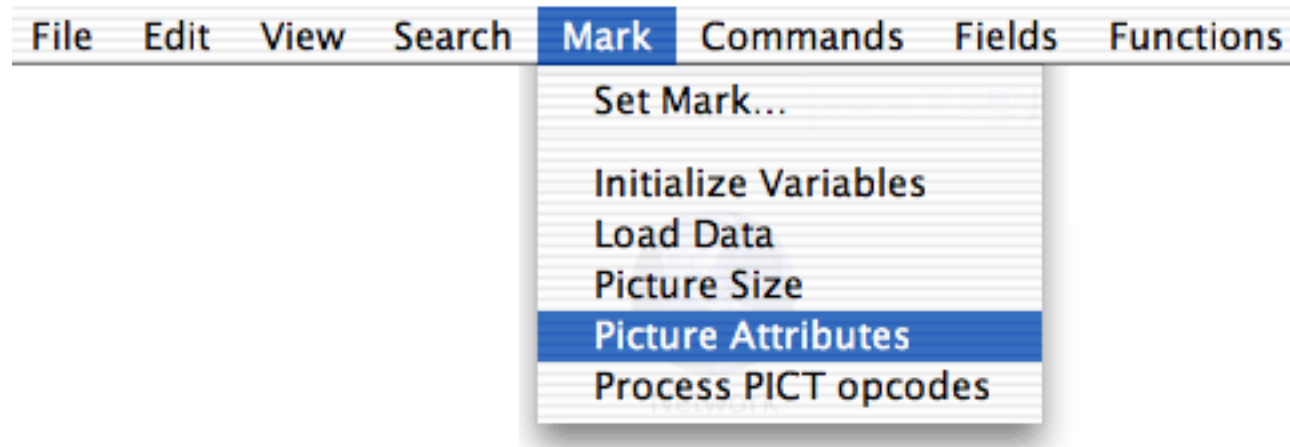
```

“Commenting Out” Statements

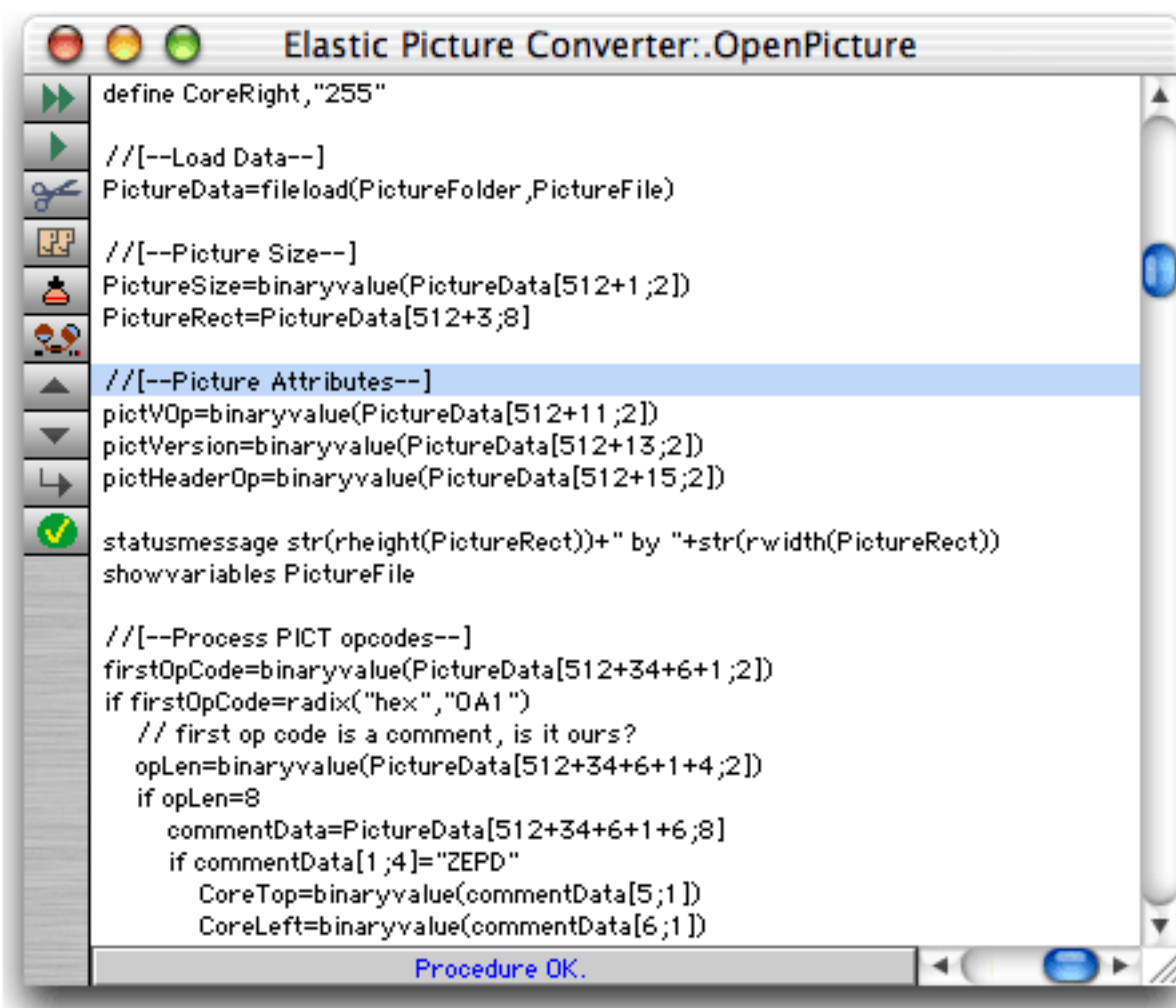
One handy use for `/* ... */` comments is to temporarily remove one or more statements from your program (usually for testing purposes). Simply put `/*` and `*/` around the statement or statements you want to remove, and those statements are effectively removed from your program without actually erasing them. (Programmers call this “commenting out” the statements, because they are temporarily “out” of the program.) To re-enable the statements simply remove the `/*` and `*/`.

Organizing Large Procedures (The Mark Menu)

A single procedure may include up to 32,000 characters of text. A procedure that long would be more than 20 pages long if printed. As a procedure grows it can be difficult to navigate within the procedure itself. The Mark menu allows you to create "bookmarks" within the procedure. These marks are listed in the Mark menu.



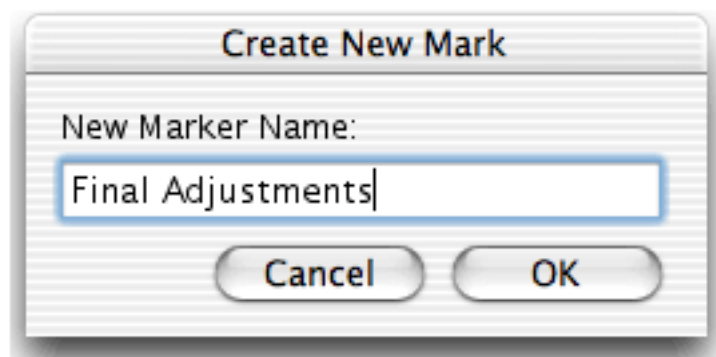
Choosing an item from this menu causes the editor to jump to the location in the procedure text corresponding to the mark.



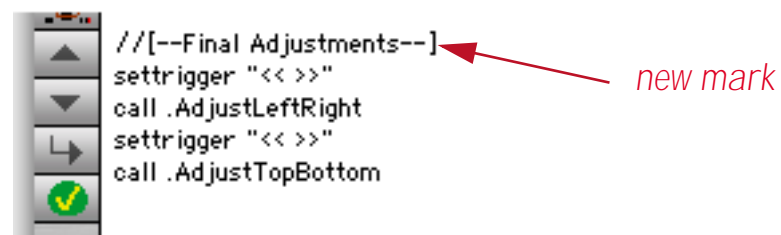
As you can see in the illustration above, a mark is simply a special comment ("[Notes To Yourself](#)" on page 304) that looks like this:

```
//[--Mark Name--]
```

You can simply type in a mark yourself, or you can use the **Set Mark...** command in the Mark menu. This command prompts you for the mark name, then inserts it into the procedure at the current location.



Here is the inserted mark.



Each procedure may contain up to 100 marks. If there are more than 100 marks then any past the first 100 will not appear in the **Mark** menu.

Suppressing Display of Text and Graphics

As a program executes the windows belonging to the database will often flicker or even redisplay over and over again as the statements are performed. Often this redisplay serves no purpose except to slow the program down and annoy you. To disable display while a sequence of steps is performed you must bracket the steps with the `noshow` and `endnoshow` statements, like this.

```
noshow
  statement
  statement
  statement
  ...
endnoshow
```

The `noshow` statement tells Panorama to suppress all display of text and graphics by the following statements. For example the `sort` and `formulafill` statements normally cause some or all of the window to be redisplayed. But if these statements follow a `noshow` statement the window will not redisplay. The `endnoshow` statement cancels the effect of the `noshow` statement and resumes normal display operation.

Note: The `noshow` statement suppresses all display that results from changes to the database. It does not, however, suppress display that is caused by changes in the configuration of database windows. For example if a procedure moves a window to the front with the `window` statement the newly visible section of the window will always be displayed, with or without a `noshow` statement. This is true for any statement that changes the window configuration: opening or closing a window, changing the size of a window, or changing the stacking order of the windows.

Updating the Display After (or Within) a NoShow Block

The `noshow` statement is great for making procedures run faster without unnecessary window updating. There's just one problem though - since the window is not updated, it winds up being wrong! Consider the procedure below.

```
noshow
  field Date
  groupup by month
  field Category
  groupup
  field Amount
  total
  outlinelevel 2
endnoshow
```

Without the `noshow` statement this procedure will cause the window to update four times. But with the `noshow` statement the final result is not displayed! To fix this you must add one of the seven statements in the table below.

Statement	Parameters	Description
<code>showpage</code>	none	Displays the entire database.
<code>showline</code>	none	Displays the current record.
<code>showfields</code>	list of fields	Displays the specified fields in the current record
<code>showvariables</code>	list of variables	Displays the specified variables
<code>showcolumns</code>	list of fields	Displays the specified fields in all visible records
<code>showrecordcounter</code>	none	Displays the number of records
<code>showother</code>	field,option	Depends on option, see documentation below

Using the **showpage** statement we can fix the program listed earlier so that it displays the final result at the end of the procedure.

```
noshow
  field Date
  groupup by month
  field Category
  groupup
  field Amount
  total
  outlinelevel 2
  showpage
endnoshow
```

The seven display statements are described in the following sections.

ShowPage

The **showpage** statement forces Panorama to redisplay all windows in the current database. Here is an example that performs several operations on the current database, but only updates the display once:

```
noshow
  field Date
  groupup by month
  field Category
  groupup
  field Amount
  total
  outlinelevel 2
  showpage
endnoshow
```

ShowLine

The **showline** statement forces Panorama to redisplay the current record in all windows in the current database. The example below clears the current record in the database, but doesn't display anything until it is completely finished.

```
noshow
  field array(dbinfo("fields",""),1,1) /* go to first field */
  loop
    clearcell
    right
  until stopped
  showline
endnoshow
```

Without the **noshow** statement you would be able to watch as Panorama cleared each cell in the line. With the **noshow** statement all of the cells will appear to disappear simultaneously.

ShowFields field,field,...,field

The **showfields** statement forces Panorama to redisplay the specified fields in all windows in the current database. You may list as many fields as you want to display, with each field separated by a comma. (If you want to display all the fields it is easier to use the **showline** statement.) The example below modifies three fields but only displays the change made to the **Balance** field.

```
noshow
  Date=today()
  Time=now()
  Balance=Credit-Debit
  showfields Balance
endnoshow
```

ShowColumns field,field,...,field

The **showcolumns** statement forces Panorama to redisplay the specified fields in all windows in the current database. In a data sheet or view-as-list window the entire column is re-displayed, not just the current record. You may list as many fields as you want to display, with each field separated by a comma. (If you want to display all the fields it is easier to use the **showpage** statement.) The example below performs two calculations on the **Balance** field, but only redisplay the column a single time.

```
noshow
  field Balance
  Balance=Credit-Debit
  RunningTotal
  showcolumns Balance
endnoshow
```

ShowVariables var,var,...,var

The **showvariables** statement forces Panorama to redisplay the specified variables in all windows in the current database. You may list as many variables as you want to display, with each variables separated by a comma. The example below adds a new record to the database without changing the display, but does show the new record count.

```
noshow
  global myCount
  addrecord
  myCount=info("total")
  showvariables myCount
endnoshow
```

Warning: The **showvariables** statement is **always** required if you want to display changes to one or more variables, even if you are not using the **noshow** statement.

ShowRecordCounter

The **showrecordcounter** statement forces Panorama to redisplay the record count in all windows in the current database. The example below adds three new records to the database but only updates the display once.

```
noshow
  addrecord
  addrecord
  addrecord
  showpage
  showrecordcounter
endnoshow
```

ShowOther field,code

The `showother` statement forces Panorama to redisplay all windows in the current database. You specify a code that tells Panorama what portion of the window to update. This is the code that Panorama uses internally, so if Panorama becomes capable of a new mode of redisplay it will automatically become available. Some codes allow you to specify a specific field to update, you can use the field name or use `All` to specify all fields. The available codes are listed in this table.

Code	Action
0	Display current cell (should use <code>showfields</code> instead)
1	Display entire page (should use <code>showpage</code> instead)
2	Cursor moved, update data sheet
3	Cursor moved, data sheet already updated
4	Update window after insertline (data sheet or view-as-list)
5	Move cursor up/down (for example after a search)
6	New line with cursor move
96	Display the tool palette
97	Display record count (use <code>showrecordcounter</code> instead)
98	Display data sheet field header (after changing field name)
99	Display after database redesign (insert field, etc.)

We recommend that you avoid this command if one of the other show commands will do the job for you.

Checking NoShow Status




Sometimes a procedure may need to check whether the display is currently disabled with `noshow`. This can be done with the `info("noshow")` function. This function returns true if `noshow` is currently in effect, false if display is normal.

```
local nstate
nstate=info("noshow")
noshow
sortup
if nstate=false()
  showpage
  endnoshow
endif
rtn
```

The example above is a subroutine that sorts the database without updating the display. It checks to see if `noshow` was already on (perhaps the procedure that called this subroutine had already turned it on), and if so, leaves it on when it finishes. The calling procedure can then continue to perform other operations without updating the display. But if the calling procedure had not enabled `noshow` before calling this subroutine the subroutine turns the display back to normal before returning (with `endnoshow`).

Disabling the Watch Cursor

As a procedure runs the cursor often flips from the arrow into a watch, or sometimes a pie chart. This helps let the user know that they may need to wait.

Arrow	Watch	Pie Chart
		

In some cases Panorama flips to the watch or pie chart cursor when it is not really necessary (especially on today's faster machines). If you want the mouse cursor to remain as an arrow while your procedure runs you can use the `nowatchcursor` statement (see "[NOWATCHCURSOR](#)" on page 5549 of the *Panorama Reference*). Here is a procedure that opens a database. This would normally cause the watch cursor to be displayed, but in this case the arrow remains active.

```
nowatchcursor
openfile "Reference Data"
watchcursor
```

The final statement re-enables the watch and pie chart mouse cursors. In this example the `watchcursor` statement isn't really necessary because Panorama always automatically re-enables these cursors at the end of any procedure.

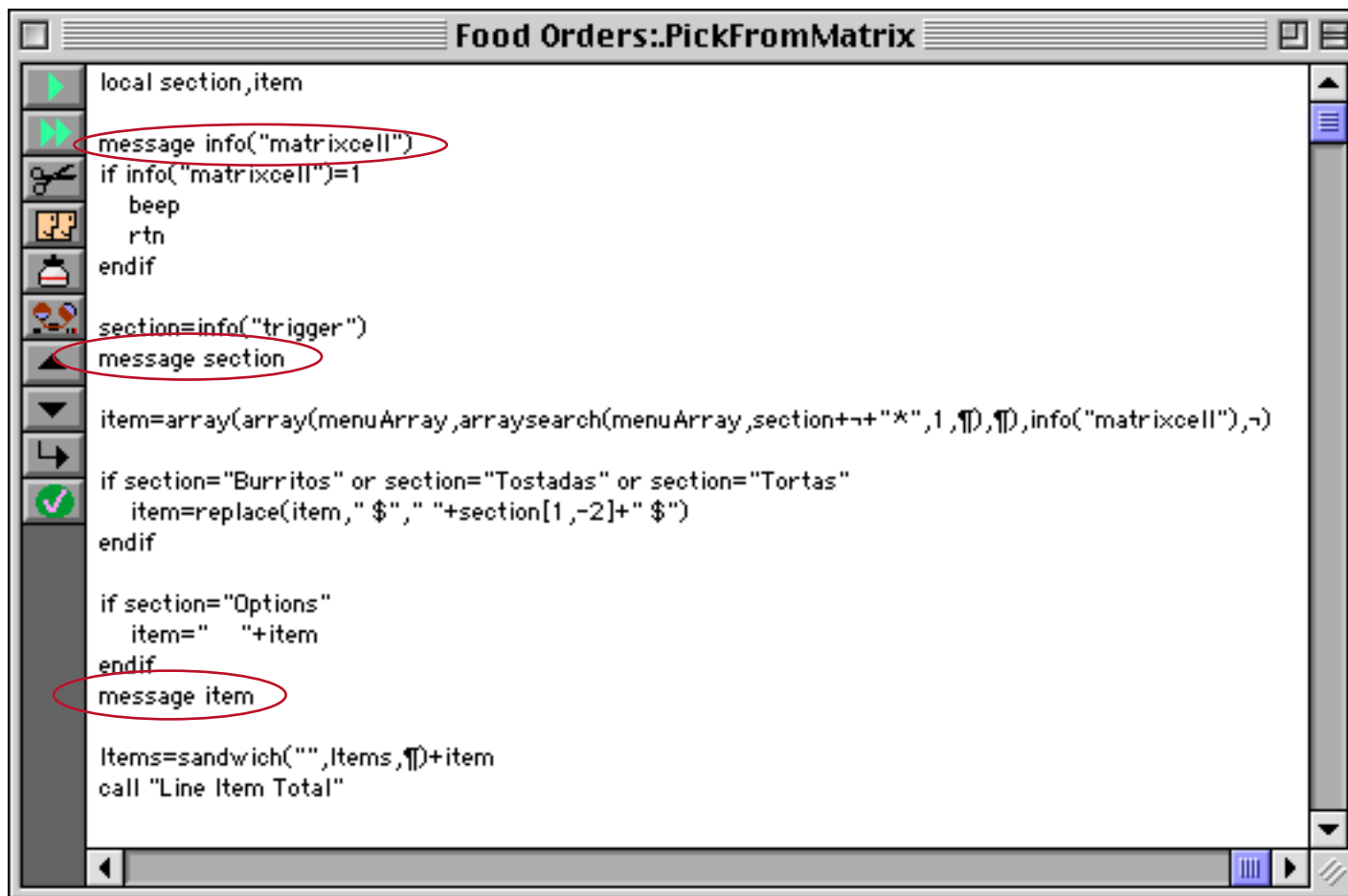
Hide and Show

Previous versions of Panorama (up to 3.0) include `hide` and `show` commands that allowed a programmer to turn off the display of text and graphics while the procedure was running. Unfortunately these commands did not give accurate control over the display, and worse, they could even crash if you attempted to use them across multiple windows. These commands are still available to retain compatibility with old databases, but we recommend that you avoid them for new applications.

Debugging a Procedure

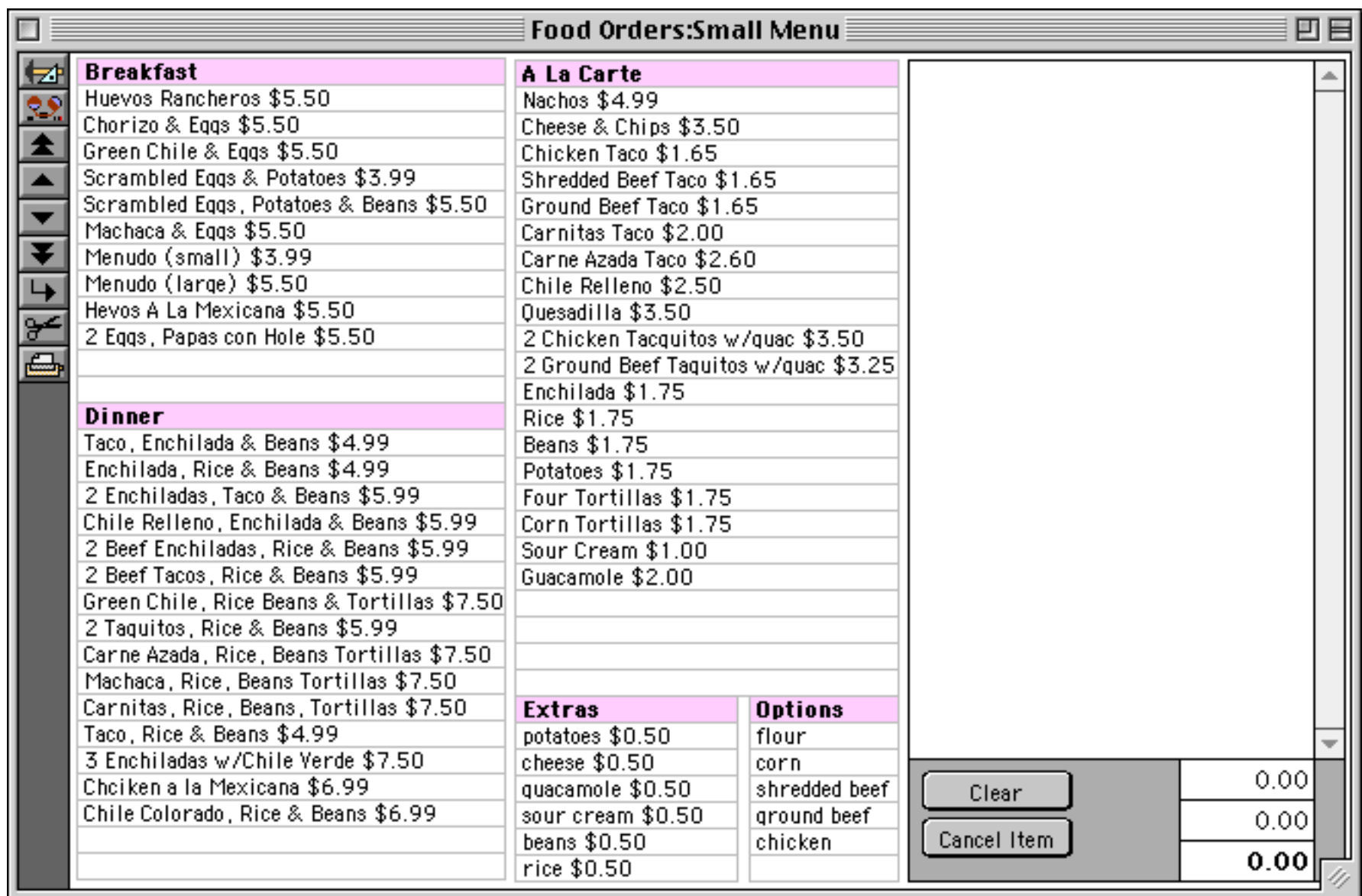
In the real world, programs often don't work correctly the first time. (Sometimes they don't work the second or the third time, either!) Panorama has a number of tools that you can use to help locate and correct the problem in a procedure.

One of the most basic tools you can use is the `message` statement. By inserting this statement at various points in your program you can display intermediate results and get a feel for what is happening in your program. For example, consider this procedure which has had three `message` statements inserted into it.



```
local section,item
message info("matrixcell")
if info("matrixcell")=1
  beep
  rtn
endif
section=info("trigger")
message section
item=array(array(menuArray,arraysearch(menuArray,section+~+"*",1,1),info("matrixcell"),~)
if section="Burritos" or section="Tostadas" or section="Tortas"
  item=replace(item,"$"," "+section[1,-2]+" $")
endif
if section="Options"
  item=" "+item
endif
message item
Items=sandwich("",Items,1)+item
call "Line Item Total"
```

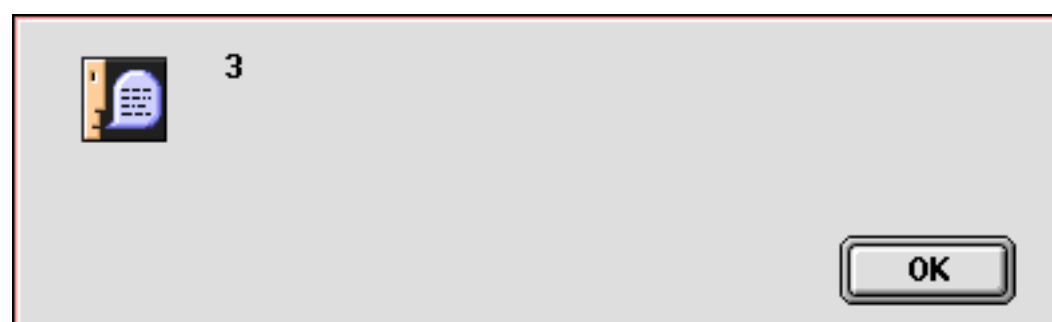

This procedure is designed to be triggered by a Matrix SuperObject (see “[Super Matrix Objects](#)” on page 939 of the *Panorama Handbook*) which contains menu items.



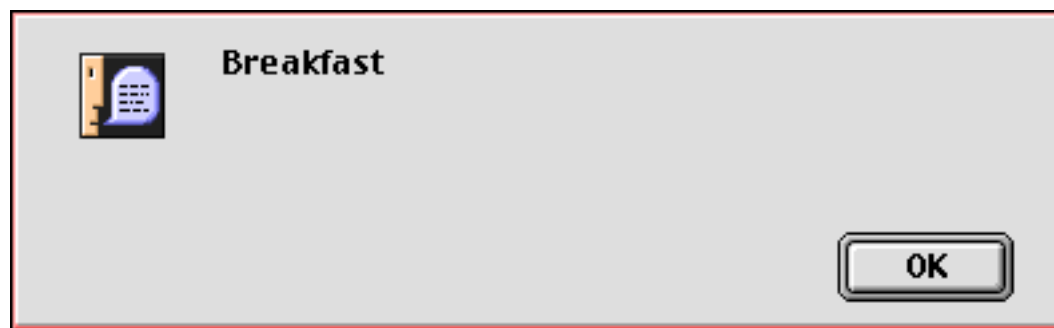
When you click on a particular matrix item the procedure is triggered.

Breakfast	
Huevos Rancheros	\$5.50
Chorizo & Eqs	\$5.50
Green Chile & Eqs	\$5.50
Scrambled Eqs & Potatoes	\$3.99
Scrambled Eqs, Potatoes & Beans	\$5.50
Machaca & Eqs	\$5.50
Menudo (small)	\$3.99
Menudo (large)	\$5.50
Hevos A La Mexicana	\$5.50
2 Eqs, Papas con Hole	\$5.50

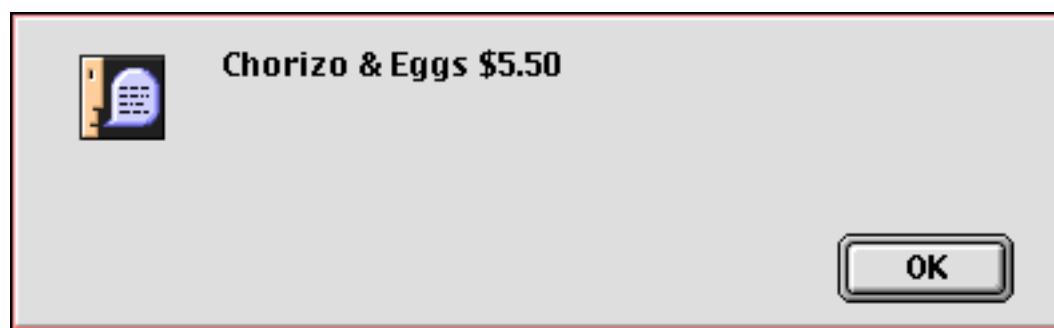
The first message statement, `message info("matrixcell")`, displays the number of the cell that was clicked on. (It also verifies that the procedure is being triggered correctly.)



The next message statement, `message section`, displays the result of the `info("trigger")` function.



The final message statement, `message item`, displays the item that has been looked up and will be added to the order. This will allow you to quickly spot any errors in the code that retrieves the item and price.



Once the procedure is working correctly you can remove the `message` statements. If you think there is any chance you might need them again you can temporarily remove them by “commenting them out” (see “[Commenting Out](#)” Statements” on page 304) like this.

 A screenshot of a code editor window titled "Food Orders::PickFromMatrix". The code is as follows:


```

local section,item
//message info("matrixcell")
if info("matrixcell")=1
  beep
  rtn
endif
section=info("trigger")
//message section
item=array(array(menuArray,arraysearch(menuArray,section+~+"*",1,1),info("matrixcell"),~)
if section="Burritos" or section="Tostadas" or section="Tortas"
  item=replace(item,"$"," "+section[1,-2]+" $")
endif
if section="Options"
  item=" "+item
endif
//message item
Items=sandwich("",Items,1)+item
call "Line Item Total"
  
```

 The code editor has a toolbar on the left with icons for running, stepping through, and other actions. The code is displayed in a monospaced font. The two `//message` statements are circled in red.

As long as the `//` is in front of the message statement the statement is disabled. Any time you want the statement back in again you simply need to remove the `//`.

Sometimes you may want to run a portion of the procedure, display a message and then stop. To do this simply add a `stop` statement after the `message` statement (see “[Stopping the Program](#)” on page 278). If both statements are on the same line then they can both be disabled with a single `//` comment.

The Panorama Interactive Debugger

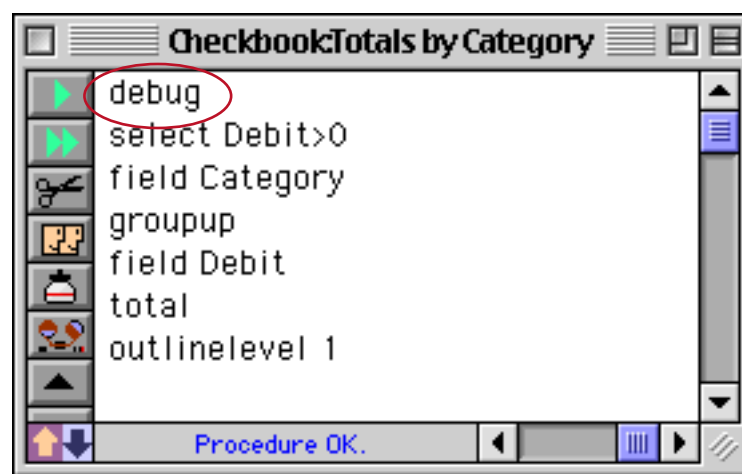
To help solve more stubborn problems Panorama includes a built in debugger. The debugger allows you to stop a procedure in the middle and execute statements one at a time (called **single-stepping**). You can actually watch as your program executes each statement, and you can check the value of fields and variables at any time.

The Debug Statement

The **debug** statement pauses the procedure so that you can examine fields and variables. You can insert a **debug** statement anywhere in a procedure, and a procedure may contain more than one **debug** statement. Usually **debug** statements are inserted into the procedure temporarily while you are getting the program running, and then removed when the procedure is operating properly.

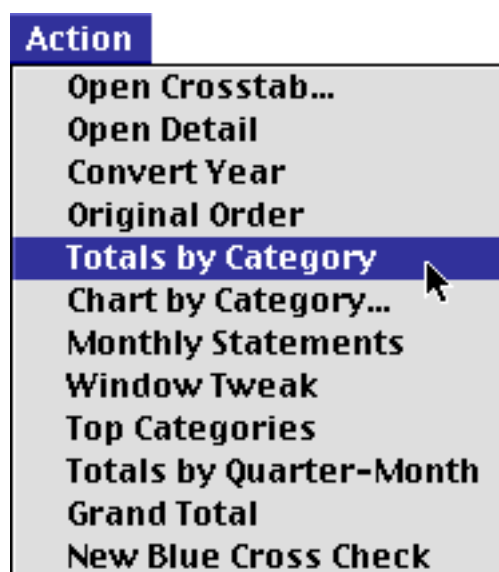
Using the Debugger

The first step in using the debugger is to add one or more **debug** statements to a procedure. In this example the **debug** statement has been inserted at the very top of the procedure, but it can be inserted anywhere.

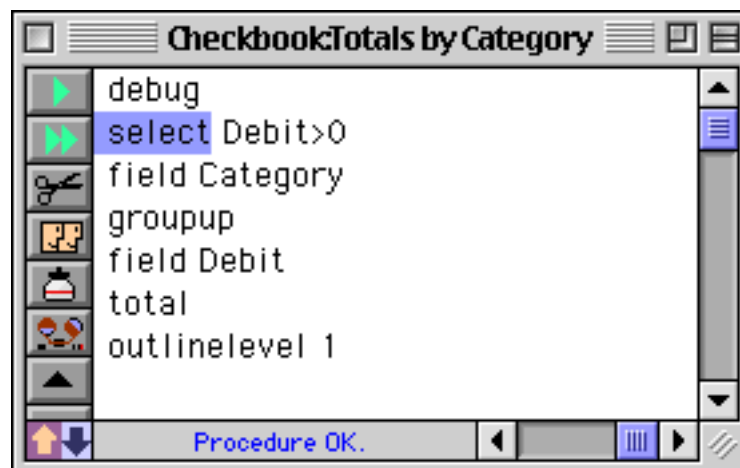


Once this is done, go to a form or data sheet window where you can test the procedure. Be sure to leave the procedure window open! If necessary, you can open an additional window in the same database - see "[Opening More Than One Window Per Database](#)" on page 169 and "[The View Wizard](#)" on page 173 of the *Panorama Handbook*.

Now start the procedure normally. Usually you will click on a button or pull down a menu item. If this is a "hidden trigger" procedure you should perform whatever action triggers the procedure.



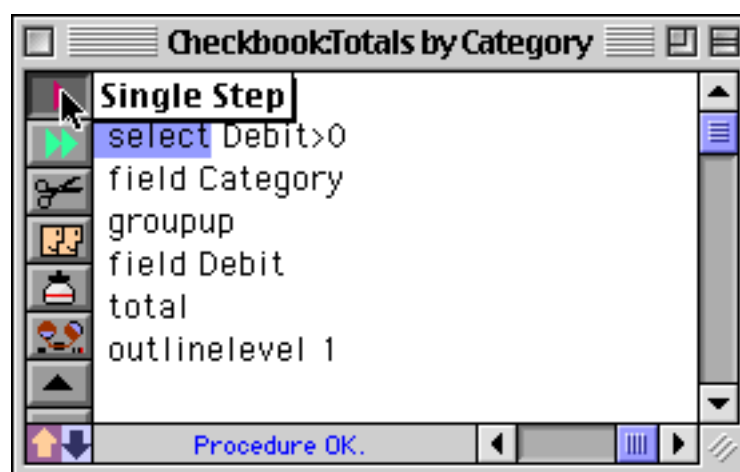
The procedure will run normally until it gets to the **debug** statement. At that point the procedure will stop and Panorama will bring the procedure window back to the front. The statement following the **debug** statement will be highlighted, indicating that it is the next statement to be performed when the procedure resumes.



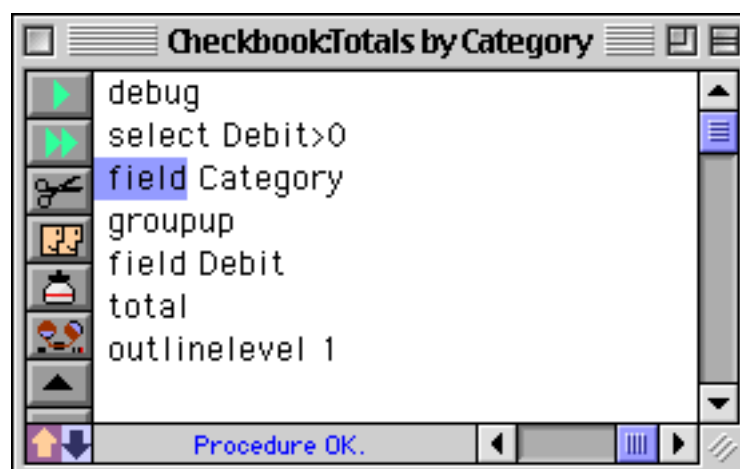
Important: If the procedure window is not open, the procedure will not stop. That's why it was so important to leave the procedure window open in the last paragraph. If you wish, you can leave **debug** statements permanently in a procedure. They won't affect the procedure unless the procedure window is open.

Single Stepping

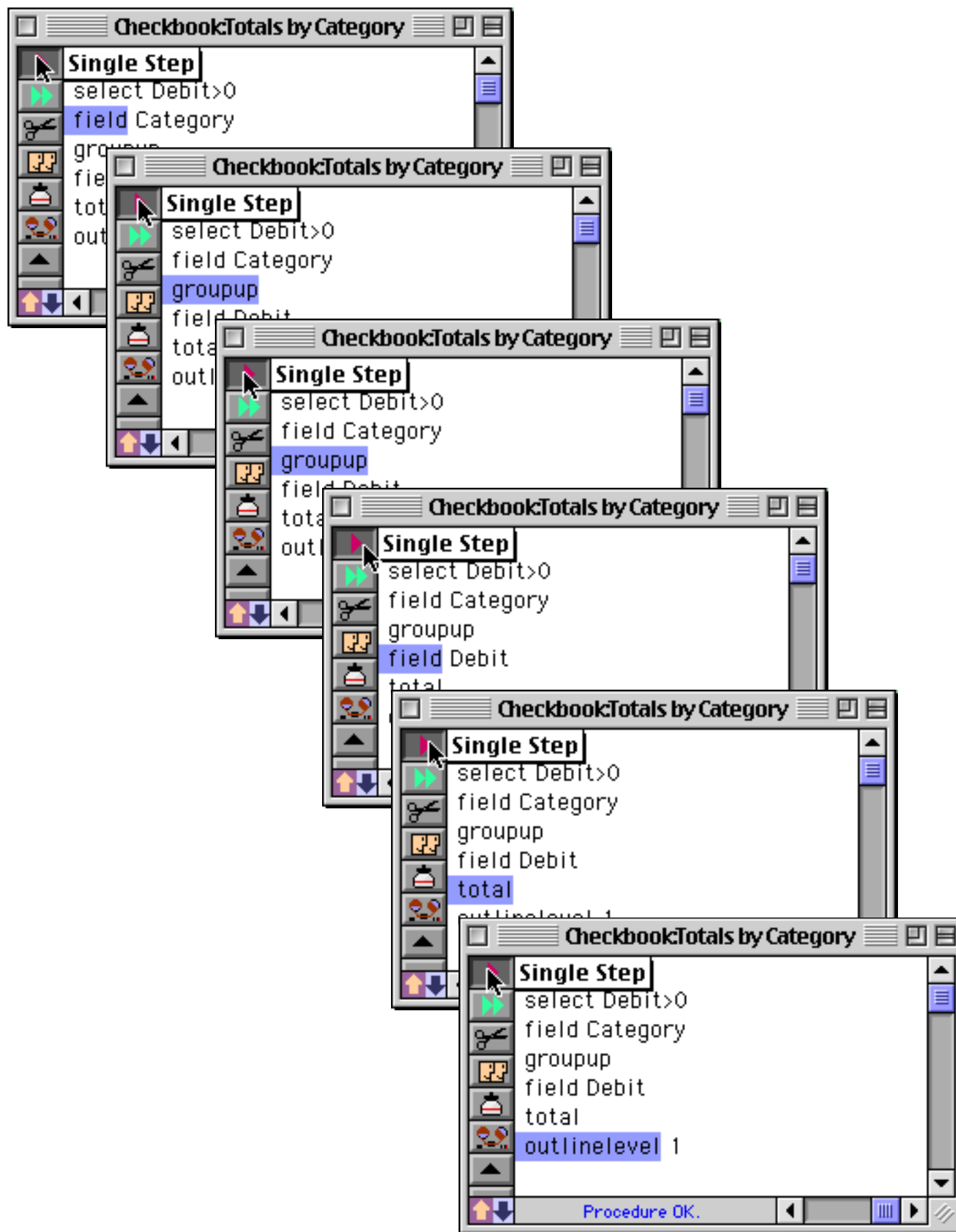
After the procedure has been stopped by a **debug** statement, you have the option of continuing the procedure one step at a time. You can watch to see what happens as each step is performed. To perform the next step, press the **Single Step** tool, or select **Single Step** from the Debug menu. (Note: you can also use the **Single Step** command without the **debug** statement simply by clicking on the tool. Panorama will start single stepping from the first line of the procedure.)



Before it performs the next statement, Panorama will move the procedure window to the back again, so that the form or data sheet (or whatever the current window was) is on top again. Panorama performs the statement, then brings the procedure window back on top again. The following statement is highlighted.



By single stepping again and again you can watch the program run. You can see as the procedure makes decisions at `if` statements, watch as a loop runs over and over again—everything your procedure does is instantly visible.



If the procedure uses `call` or `farcall` statement to trigger a subroutine (see “[Subroutines](#)” on page 261), single stepping usually considers the subroutine to be a single step. In other words, in one step Panorama will perform the entire subroutine. However, if the window containing the subroutine procedure is open, Panorama will single step through the subroutine, letting you see each step in it.

Resuming Full Speed Execution

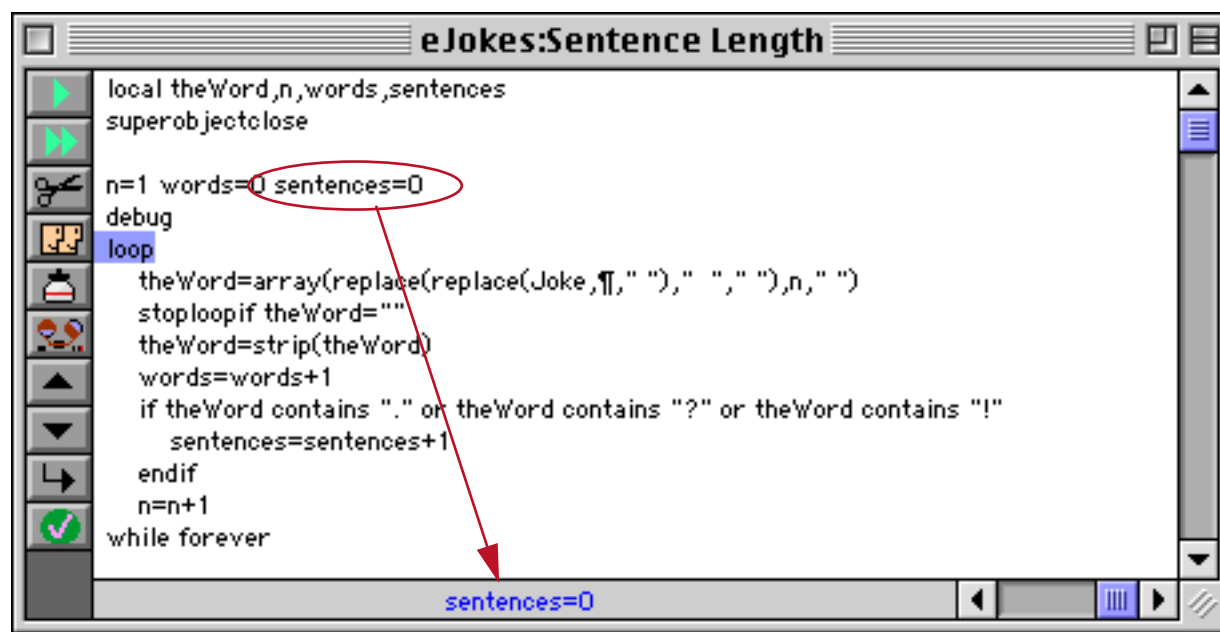
If you want the procedure to start up again at full speed, press the **Run** tool or select **Run** from the Debug menu. The procedure will start up again at full speed from the current spot. It will continue at full speed until it either reaches the end of the procedure, or it comes to another **debug** statement in an open procedure window.

Making Corrections to a Procedure

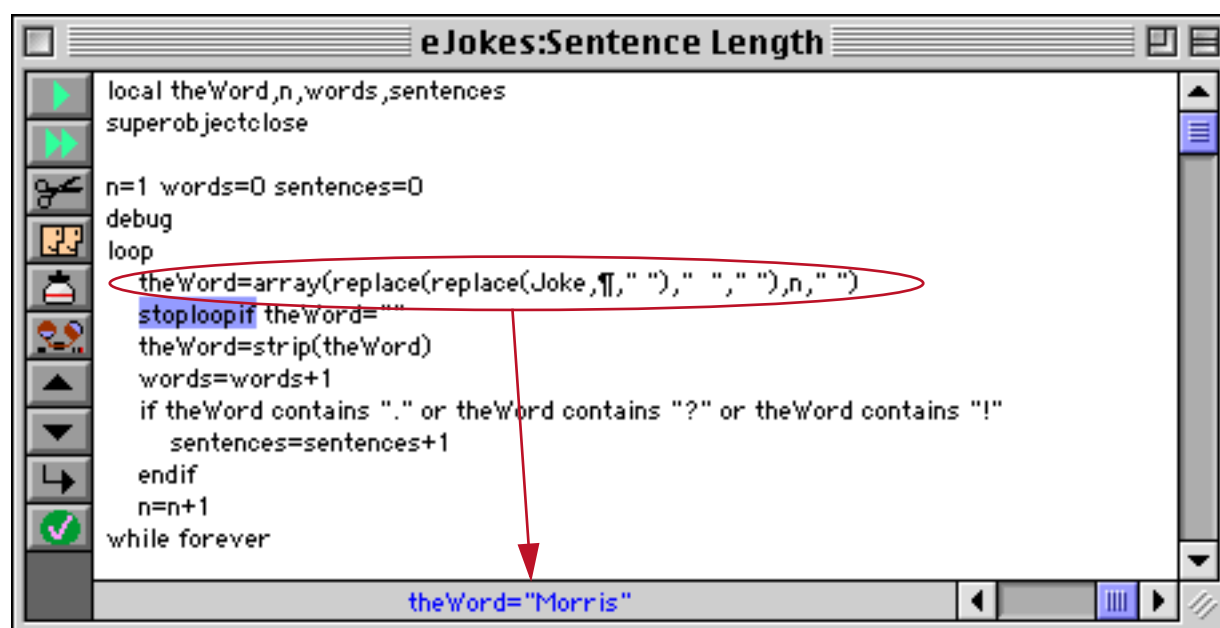
In the course of debugging you may find a problem with your procedure. To fix the problem, just edit the procedure. However, after you change the procedure you can no longer single step or continue the procedure. You must start over again from the top after any kind of change.

Watching Computations

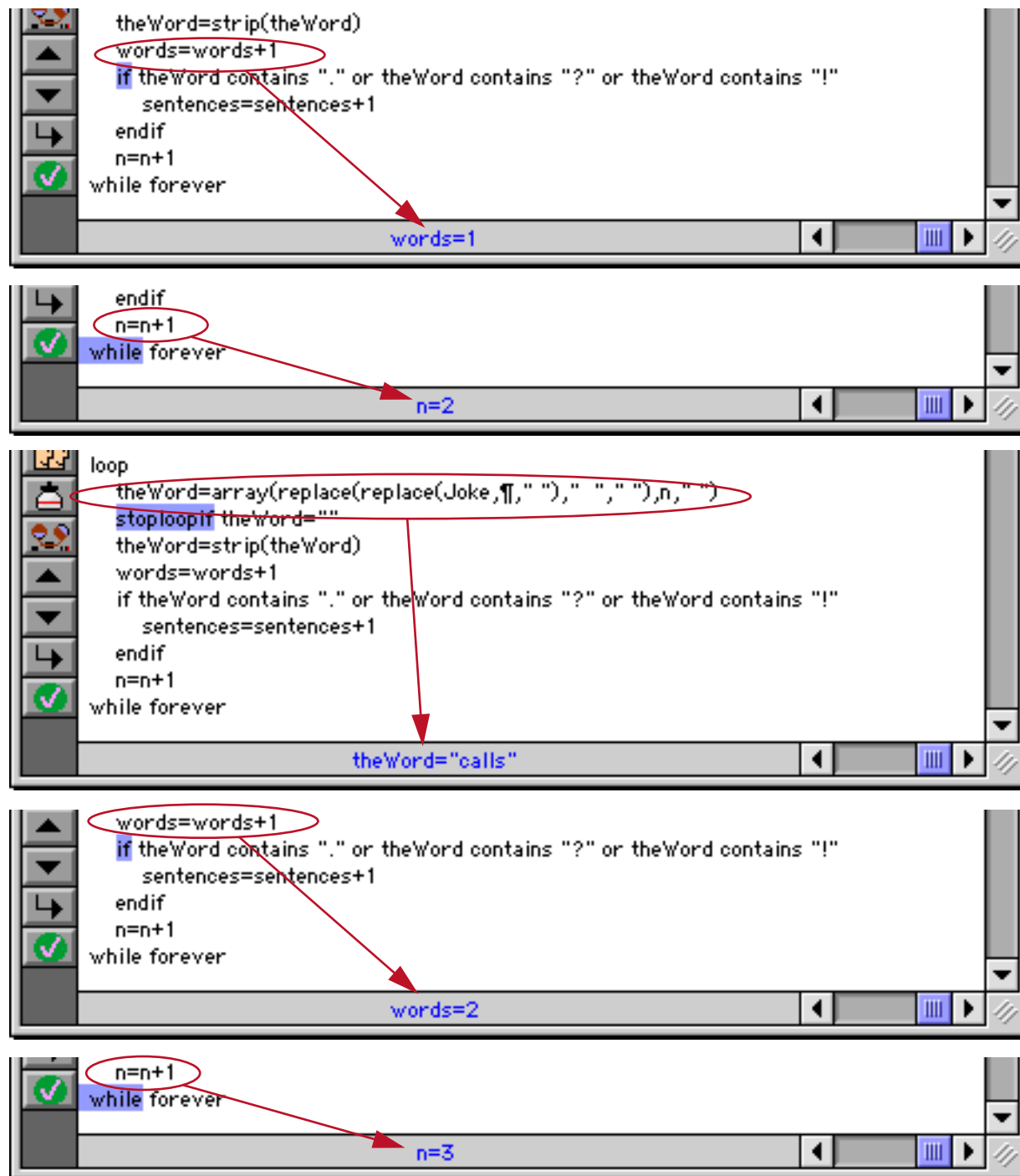
When you single step through a procedure Panorama updates the status bar to show the result of each assignment statement (see "[Assignment Statements](#)" on page 243). This makes it easier to follow along with what is going on in the procedure. For example, the procedure shown below has just stopped after the **debug** statement. The status bar shows the result of the last assignment statement, **sentences=0**.



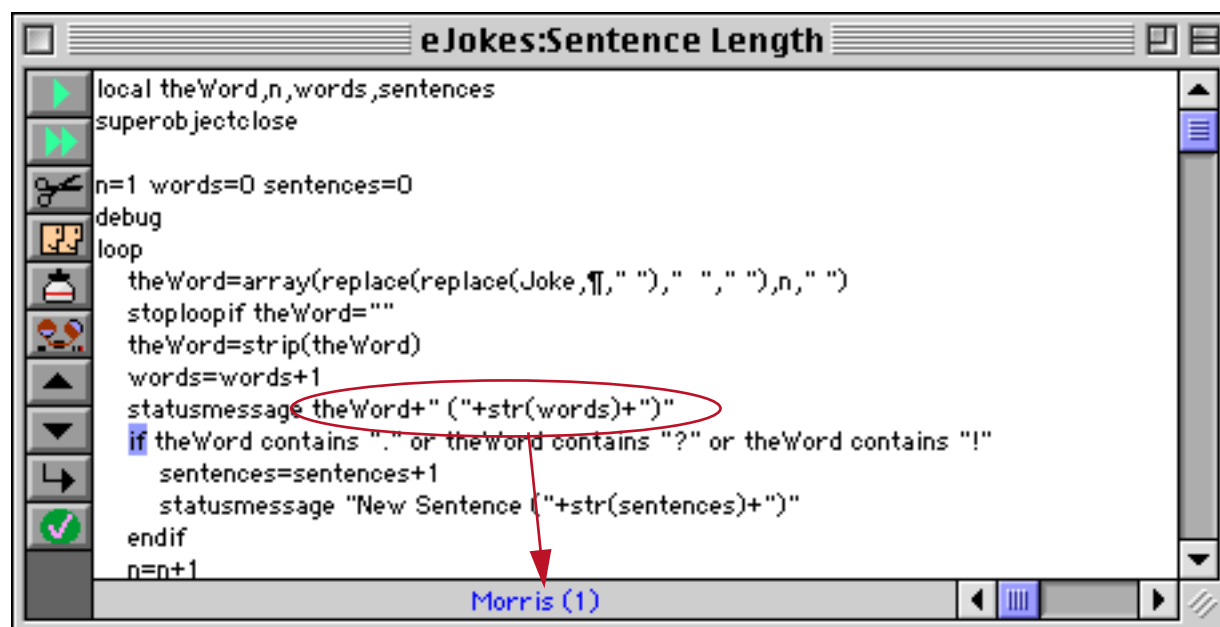
Single stepping twice executes the loop and assignment statements. Again, the status bar shows the result of the assignment, in this case the word **Morris**.



Each time you single step through an assignment statement the result is shown in the status bar.



You can use the `statusmessage` statement to display any formula in the status bar. This statement is similar to the `message` statement, but instead of displaying the message in an alert it displays it in the status bar, as shown below.



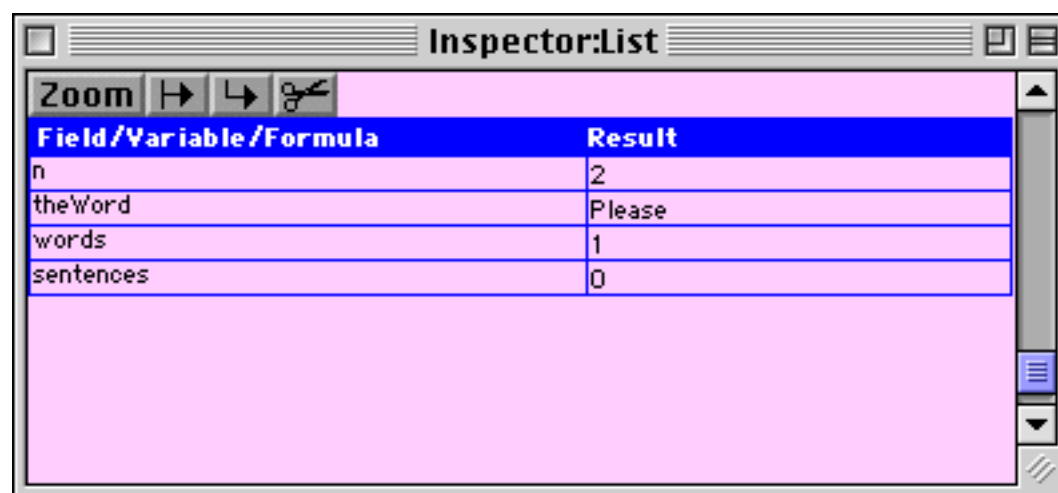
The `statusmessage` statement works any time the procedure window is open, even when the procedure is running at full speed. A strategically placed `statusmessage` statement can be ideal for watching the progress of a loop. For example the `statusmessage` statement in the procedure above will let you watch as the procedure counts the words in each sentence — 1, 2, 3 If the procedure window is closed the `statusmessage` statement is simply ignored, so it is safe to leave this statement in your final procedure in case you need it later. (When the window is open the procedure may run slower than normal due to the time taken to update the status bar.)

Using the Inspector to Examine Fields, Variables and Formulas

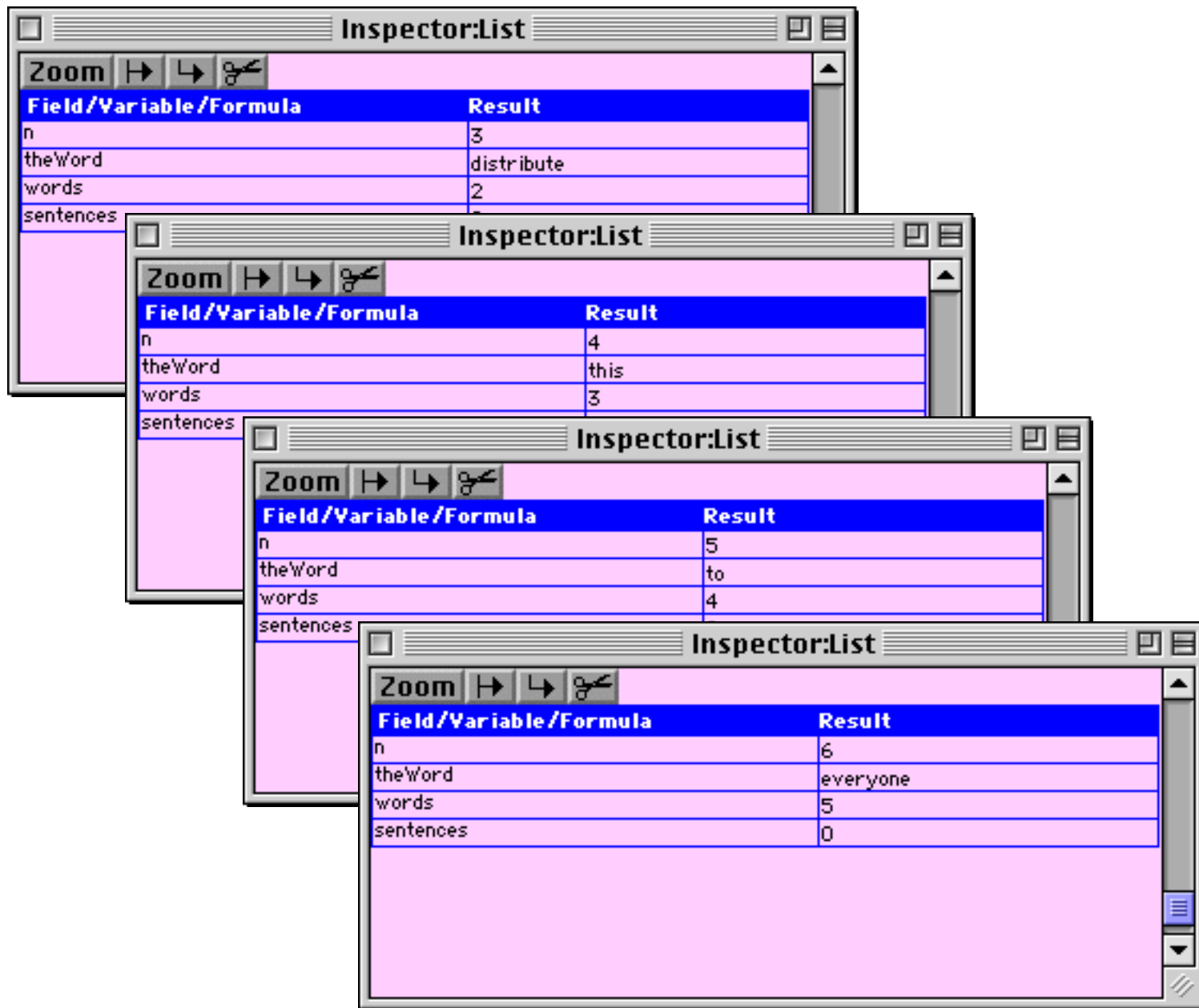
During debugging, you'll often need to examine the contents of fields and variables. If the fields and/or variables you are interested in are not already visible in a form or data sheet window, you can use the Inspector window to watch them. To access this window, choose the **Open Inspector** command in the Debug menu.



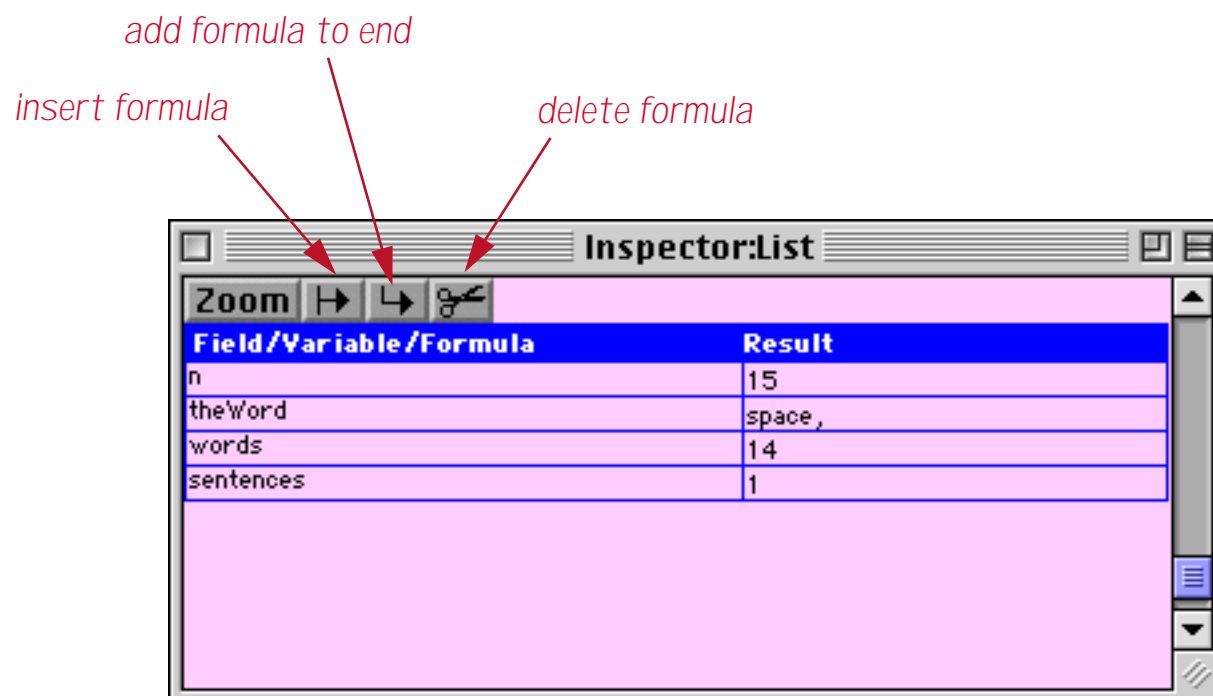
The **Inspector** window displays two columns. The left column is for formulas that you enter. The right column displays the result of each formula.



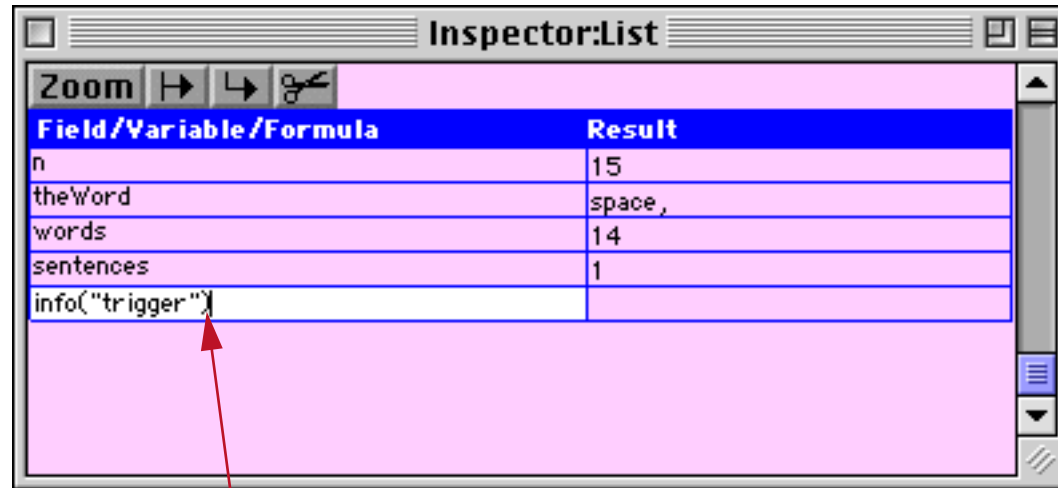
Each formula may consist of a field, a variable, or a more complex formula with fields, variables, and/or functions. The Inspector window calculates and displays the result of each formula. The calculations are updated every time a field or variable is changed, so you can actually watch the data change as you single step or proceed through the procedure.



Use the buttons at the top of the window to add and remove formulas.

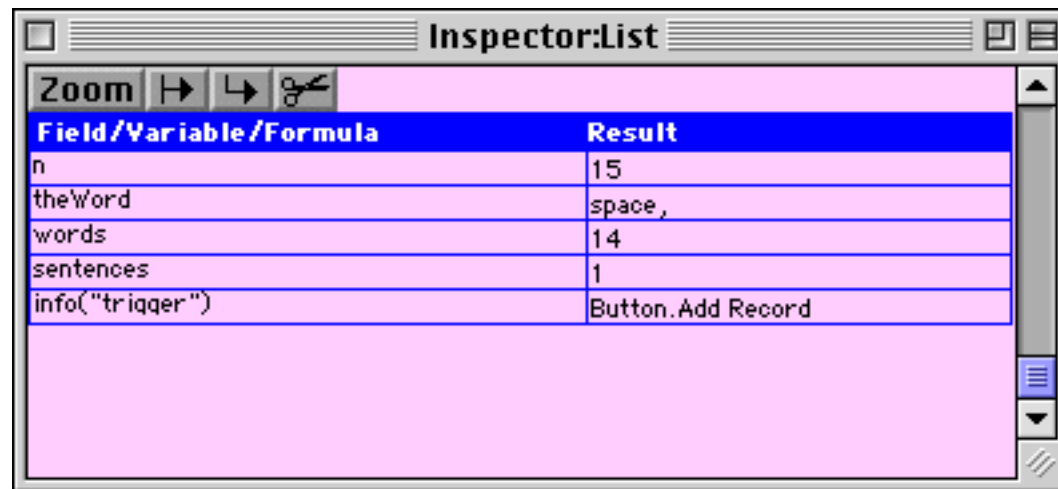


To edit a formula, click on it then begin typing.

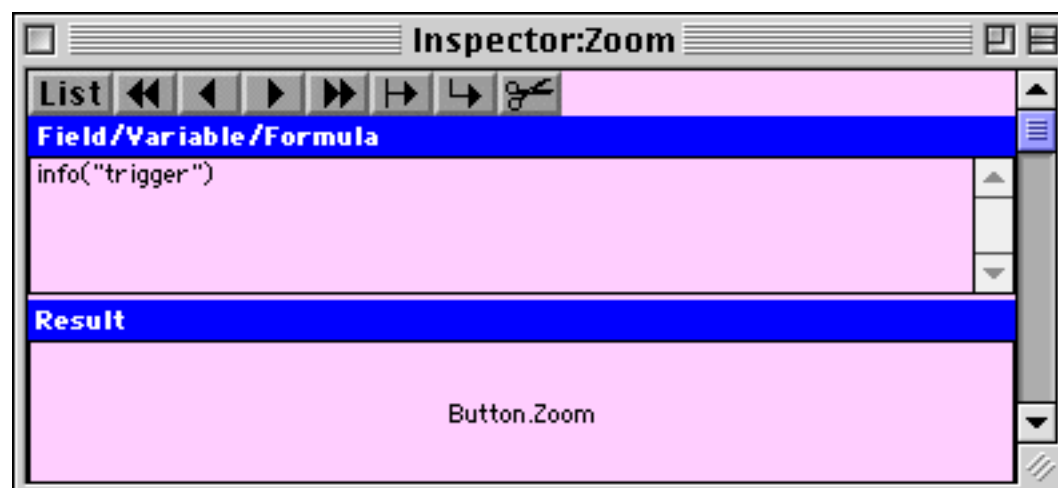


click to edit formula

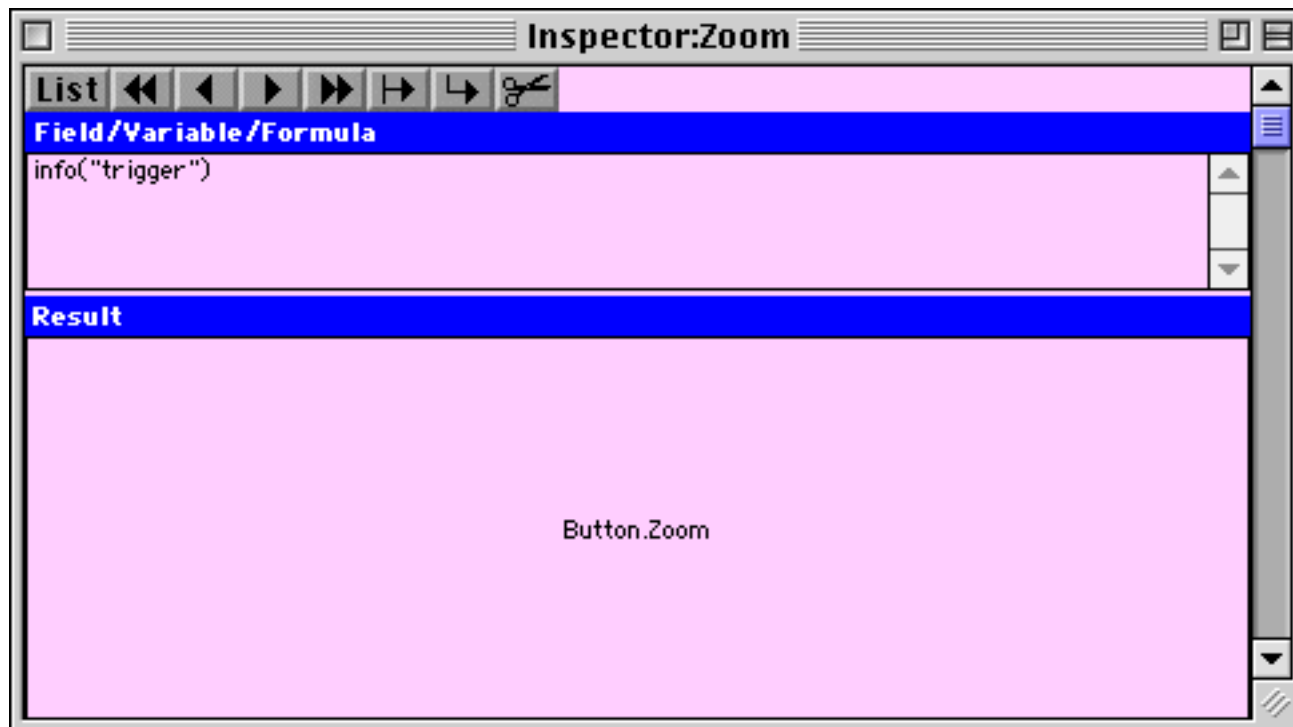
Press **Return** or **Enter** to see the result.



The **Inspector** window normally displays a list of formulas, one line per formula. If you need to display a formula or result that is more than one line high switch to the Zoom mode. In this mode only one formula and result is displayed at a time.



If necessary you can enlarge the window to display a result that doesn't fit in the normal window size.



To switch back to the list mode press the **List** button.

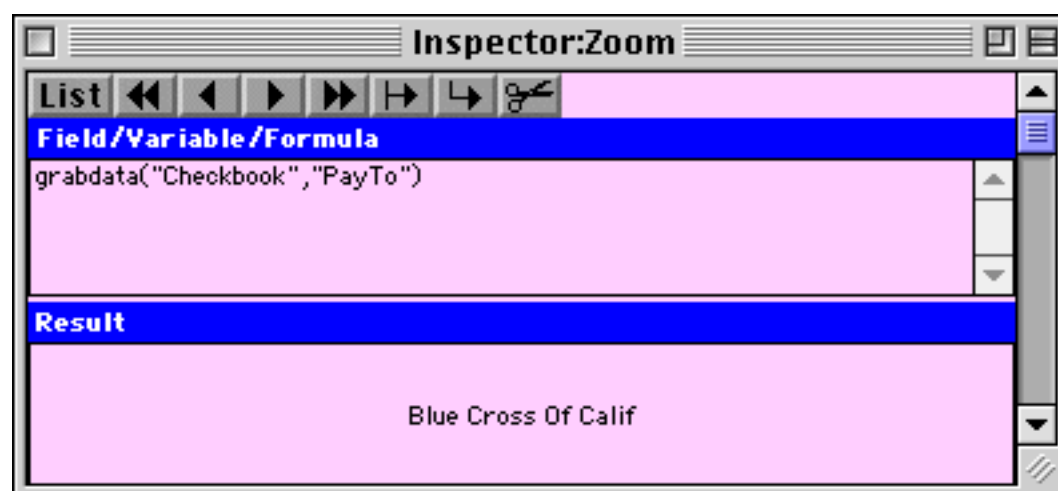
What Fields or Variables can be Displayed?

Sometimes you may enter a formula that looks correct to you, but no result appears in the **Inspector** window. Why does this error occur? This means that the field or variable is not currently accessible to the procedure being debugged. As a procedure runs, it may switch from database to database, and variables may be created and destroyed (see "[Variable Accessibility](#)" on page 250). The **Inspector** window cannot display fields or variables that the procedure cannot access.

If you are trying to display a local variable, that variable may not exist. Local variables only exist while the procedure is actually running. You can only see the contents of a local variable while the procedure is stopped in the middle or single stepping. Before the procedure is started, or after it is finished, the local variable does not exist and cannot be inspected.

When displaying fields, the **Inspector** window always displays the value of the field in the current debug database. Usually this is the database that contains the procedure. However, if the procedure switches to a different database (with the `window` or `openfile` statement), the **Inspector** window will also switch to the new database. If no result appears, you are probably attempting to display a field in another database.

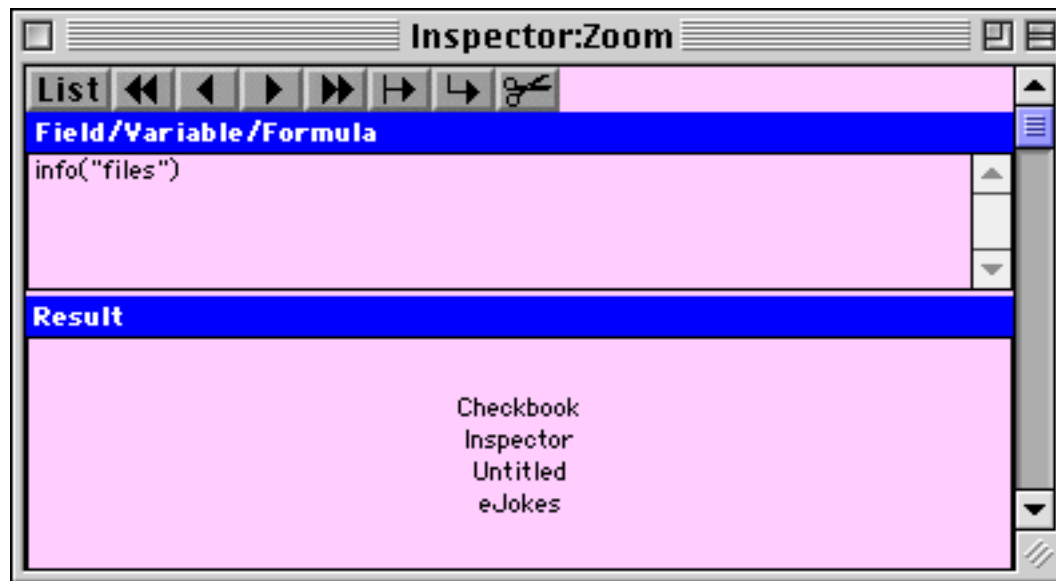
Use the `grabdata()` function to display the value of a particular field in a particular database no matter what database is being debugged. For example, to always display the `PayTo` field in the `Checkbook` database, type the formula `grabdata("Checkbook","PayTo")` into the **Inspector** window as shown below.



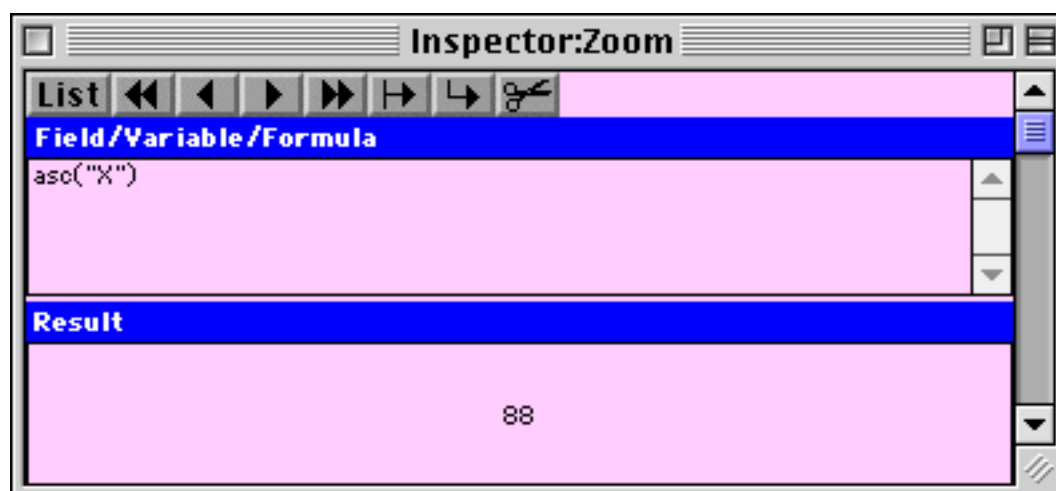
As long as the `Checkbook` database remains open in memory you'll be able to see the contents of this field.

Displaying Functions

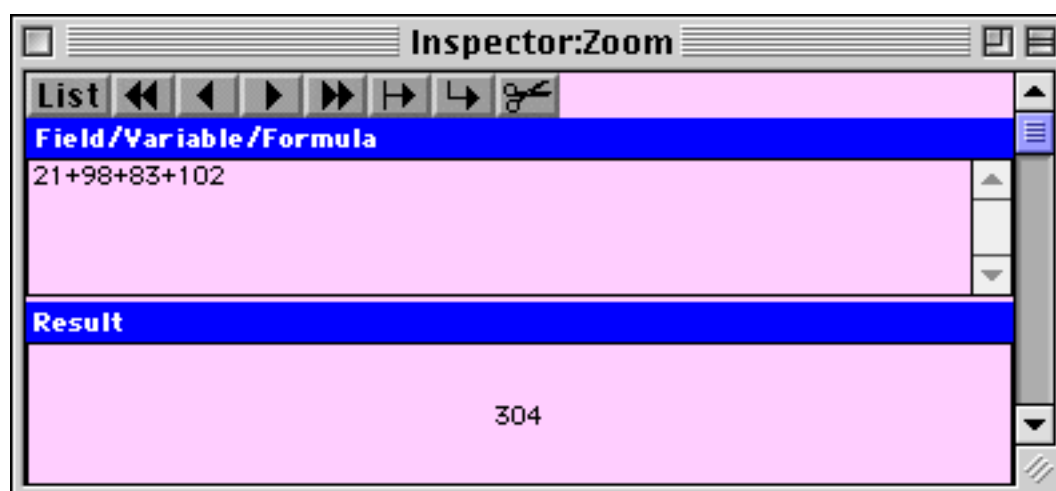
Don't forget that the **Inspector** window can display any formula, not just fields and variables. One handy application for the Inspector window is to look at the results of info(functions. For example, you can display the function `info("trigger")` to see how a procedure was triggered, or `info("files")` to see what databases are currently open.



You can use the `asc(` function to look up the ASCII value of a character (see [“Characters and ASCII Values”](#) on page 87).



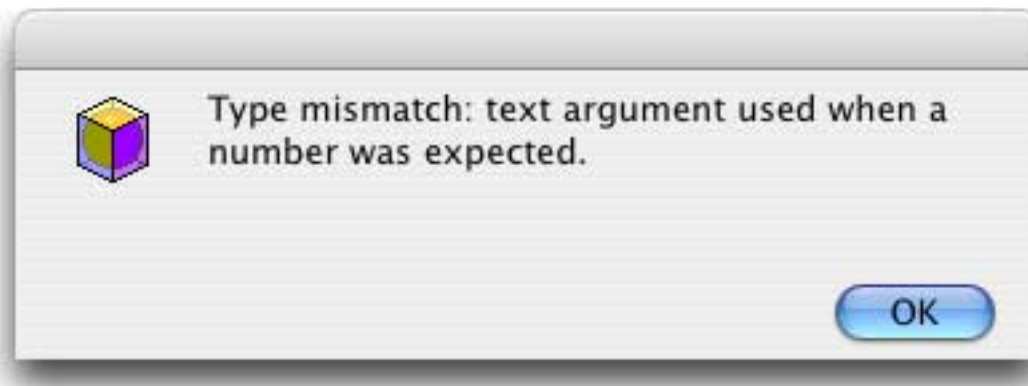
Or you can use the Inspector as a handy calculator.



Note: The **Formula Wizard** also can be used as a handy calculator. See [“Using the Formula Wizard”](#) on page 29.

Error Detail Wizard

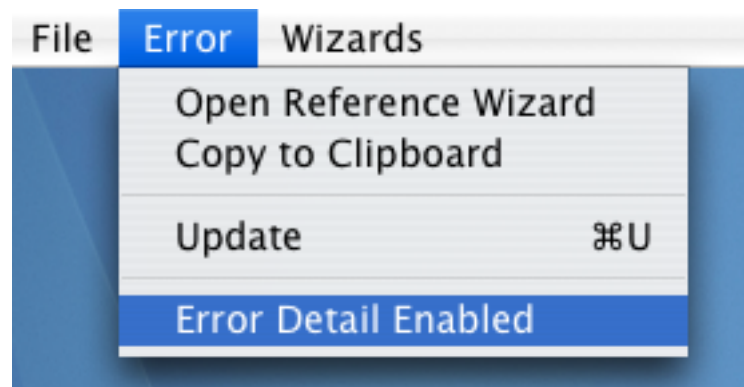
The **Error Detail** wizard can help track down the source of an error in a procedure or formula. When an error occurs, Panorama normally displays an alert, like this:



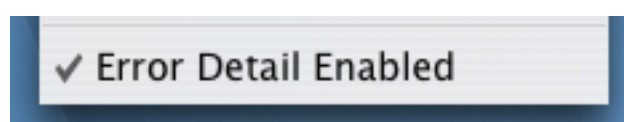
Once it is enabled the **Error Detail** wizard can give you more help in tracking down errors like this. Start by opening the wizard. As you can see, it is initially disabled.



To enable the wizard choose the **Error Detail Enabled** command in the Error menu.



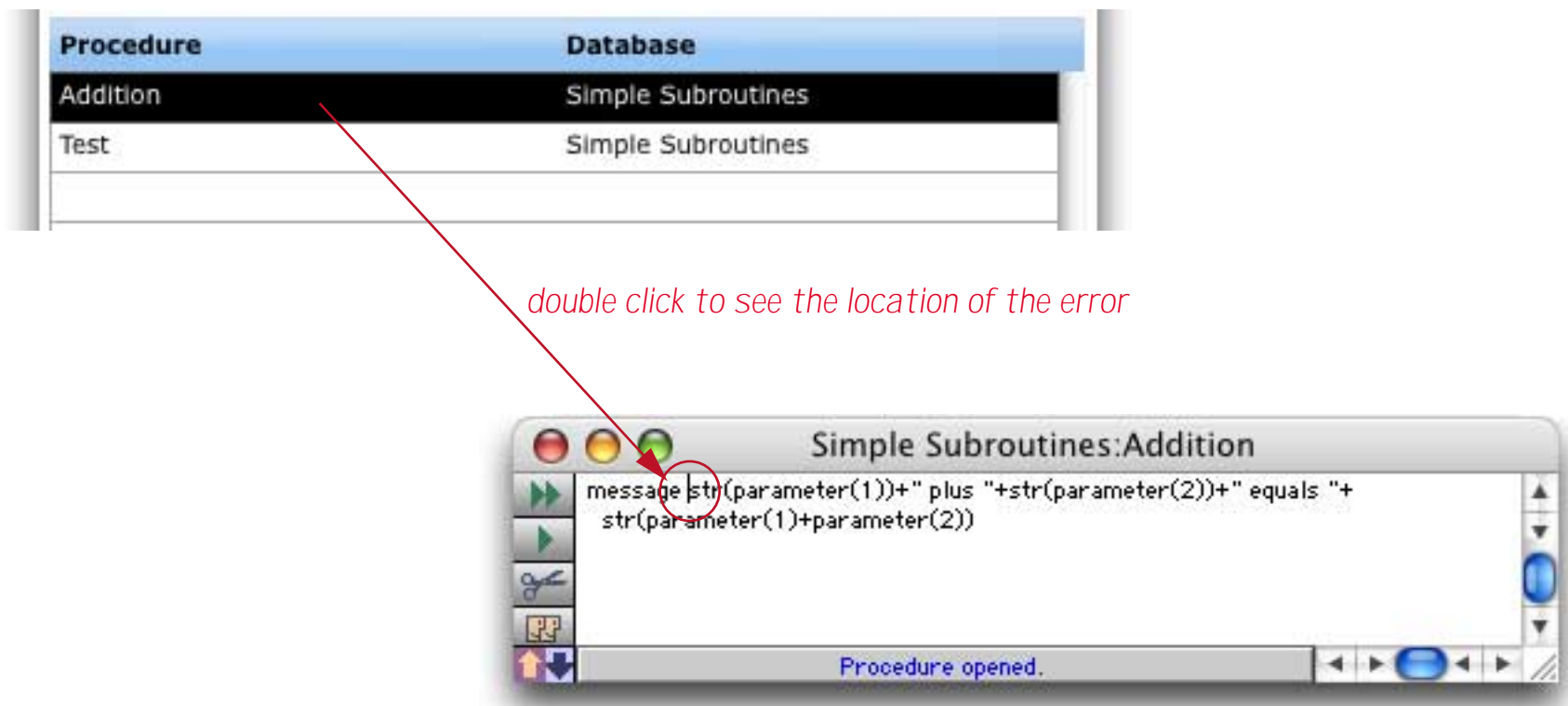
The menu always shows the current status.



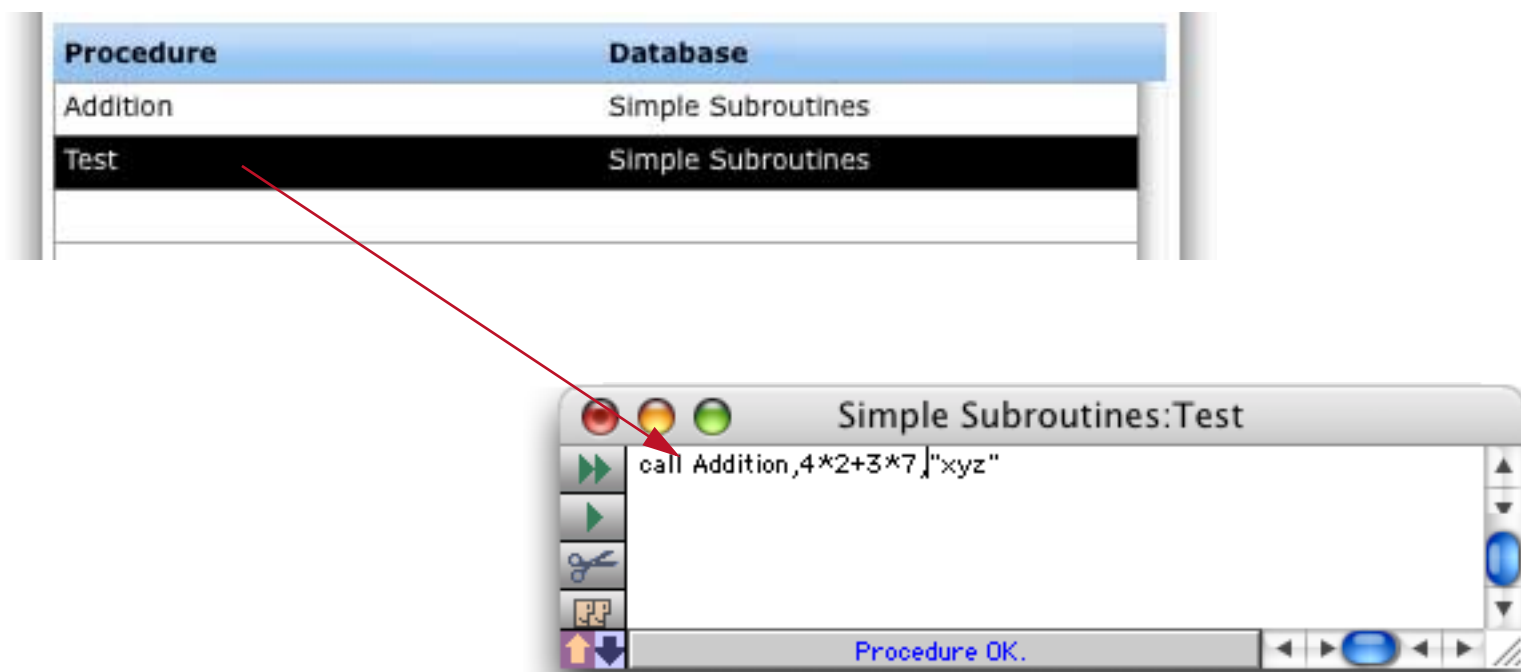
Once you enable **Error Detail** it will remain on until you explicitly turn it off (even if you close the wizard or completely quit and relaunch Panorama). Note: Sometimes Panorama becomes so confused by an error that this wizard doesn't display the correct information. Fortunately this happens quite rarely.

Finding the Source of the Error

In addition to providing more information about an error the **Error Detail** wizard can also pinpoint the exact location where the error occurred. To find the exact location double click on the procedure name.

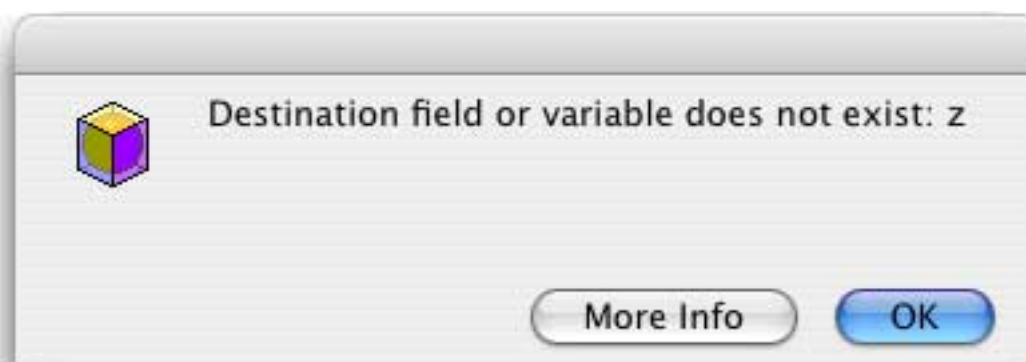


In some cases (like the example above) the actual problem isn't at the location where the error occurred, but further up the "call chain", where the procedure was called (see "[Subroutines](#)" on page 261). You can double any procedure in the "call chain" to see where the procedure containing the error was called.

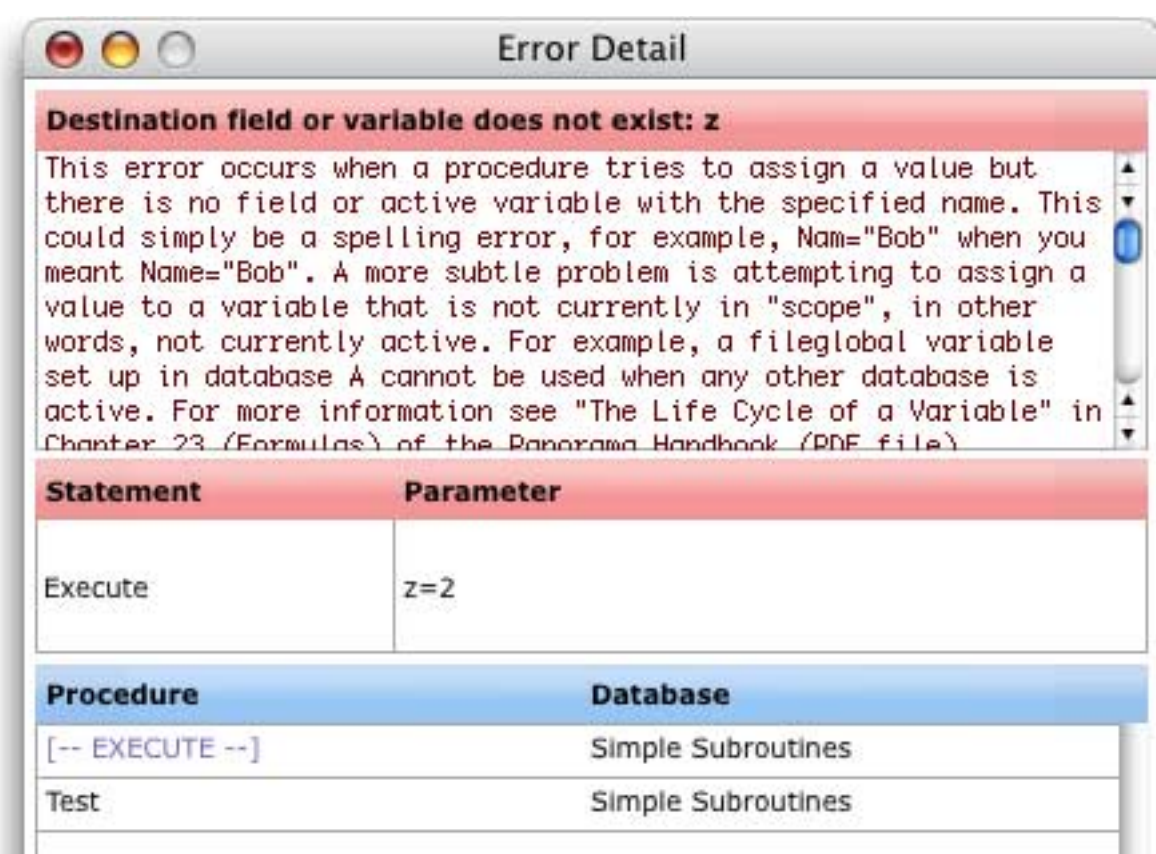


Now we can see the problem — the call statement is passing the text "xyz" when it needs to be passing a number like 123. Changing "xyz" to a number will fix the problem.

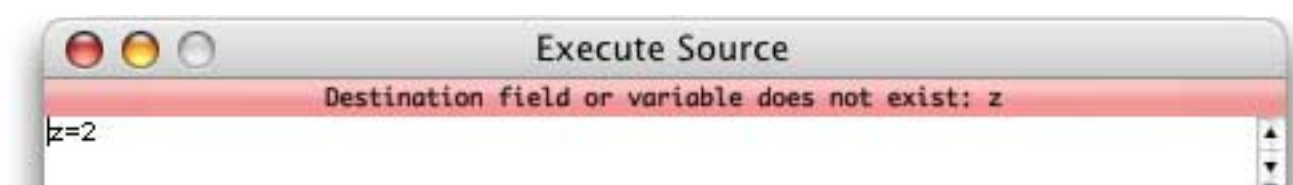
The wizard can also help track down problems that occur in an `execute` statement (see “[Building Subroutines On The Fly \(The Execute Statement\)](#)” on page 280). Suppose you see an error message like this:



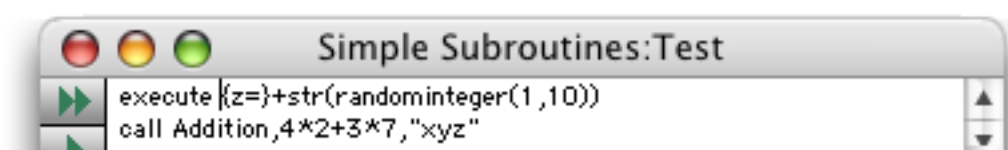
Press the **More Info** button to see the additional detail.



As you can see, the procedure containing the error has no name because it was built on the fly by an `execute` statement. Double click on this line to see the statement itself.



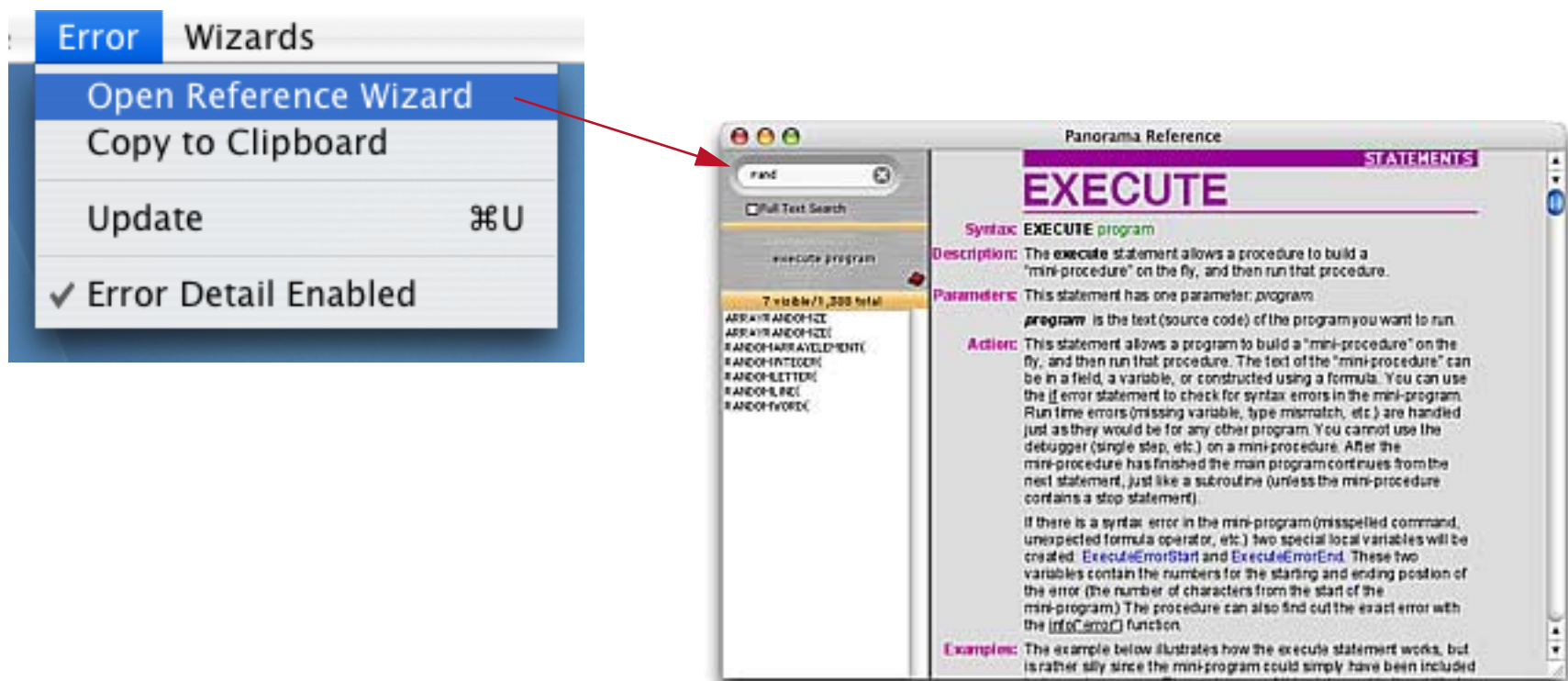
But where is this in the original program? Double click on the second line to see the procedure that contains the `execute` statement.



Ok, now the problem should be easy to fix. Notice that the actual statement in the two windows does not match. This is because the **Execute Source** window shows the statement after the formula has been evaluated. This can be very useful if the formula used to build the statement on the fly contains an error.

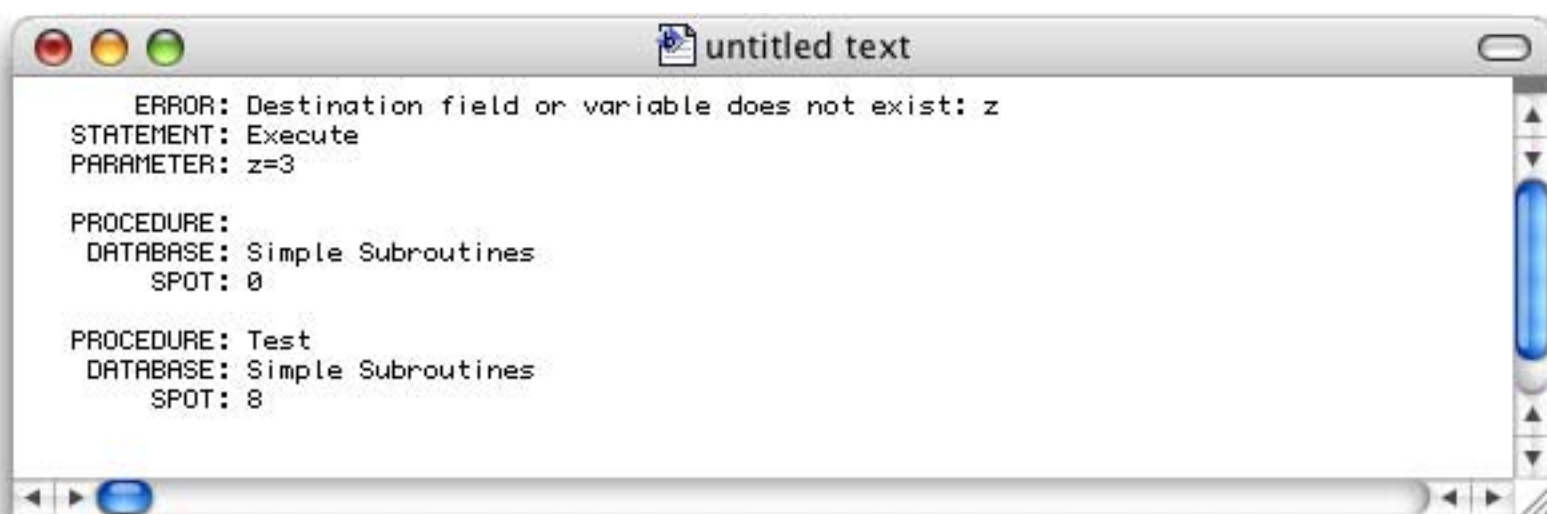
Open Reference Wizard

Need more information about the statement that the error occurred in? Simply choose **Open Reference Wizard** from the **Error** menu. The **Programming Reference** wizard (see “[Programming Reference Wizard](#)” on page 237) will automatically open and display the page for the statement in question.



Copy to Clipboard

This command copies the error detail so that it can be pasted into an e-mail, allowing it to be sent to someone else. Here's what the error detail looks like in text format.



If the PROCEDURE name is blank this code is in an **execute** statement. The SPOT indicates the location of the error within the source code. The spot is in characters, so for example the **call** statement in the test procedure is 8 characters from the start of that procedure (or in this case, the start of the statement defined by the **execute** statement).

Error Detail Problems

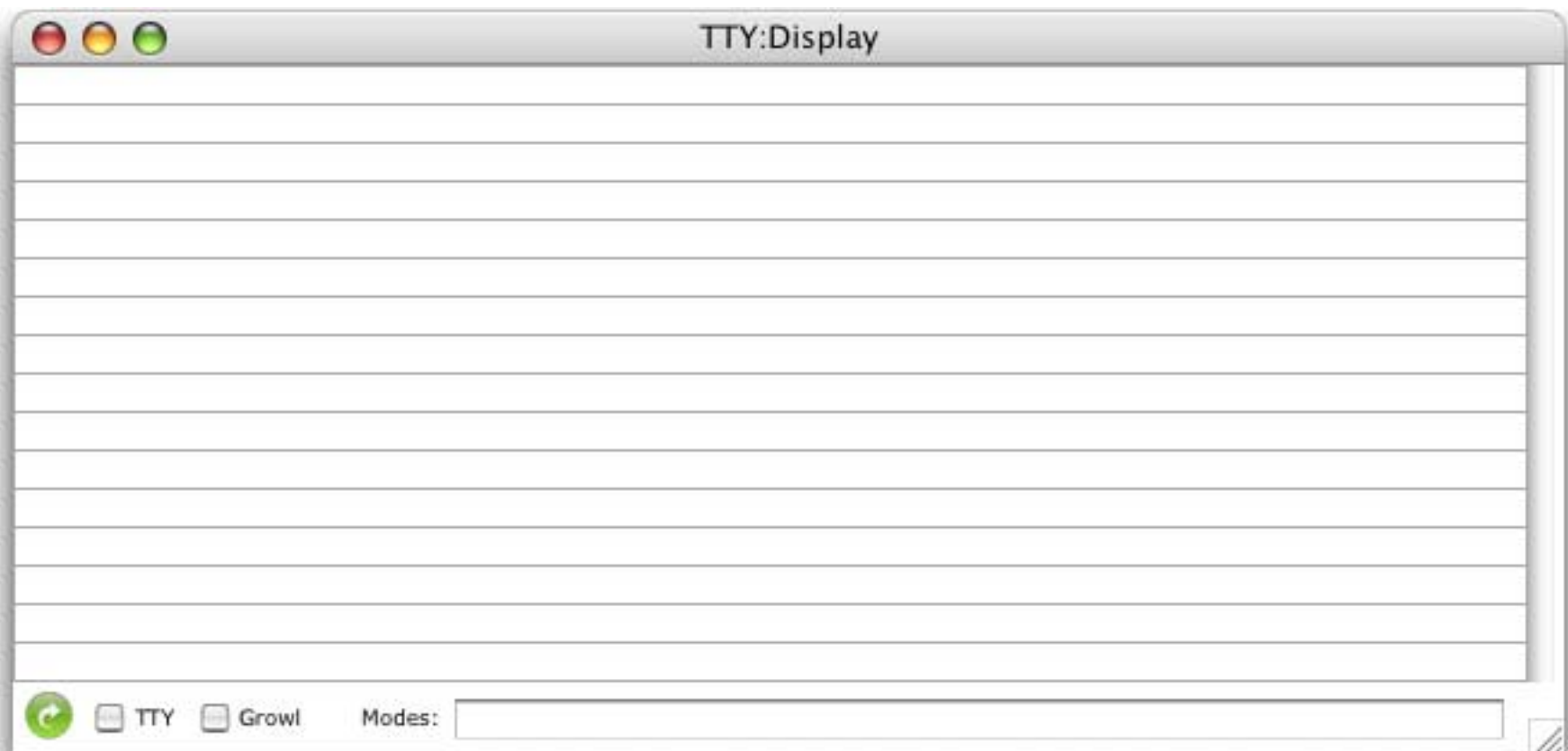
The **Error Detail** wizard works well in almost all situations, but there are a few advanced programming techniques can trip it up and prevent it from providing accurate information. Panorama was not originally designed to support this wizard, and in some situations we were simply unable to retrofit it to do so. The good news is that it will be immediately obvious when this happens, so you won't waste time tracking down bogus information. However in these cases you'll have to resort to more old-fashioned methods for tracking down the problem, for example inserting **message** statements into your code.

Debugging with the TTY (Virtual Teletype) Wizard

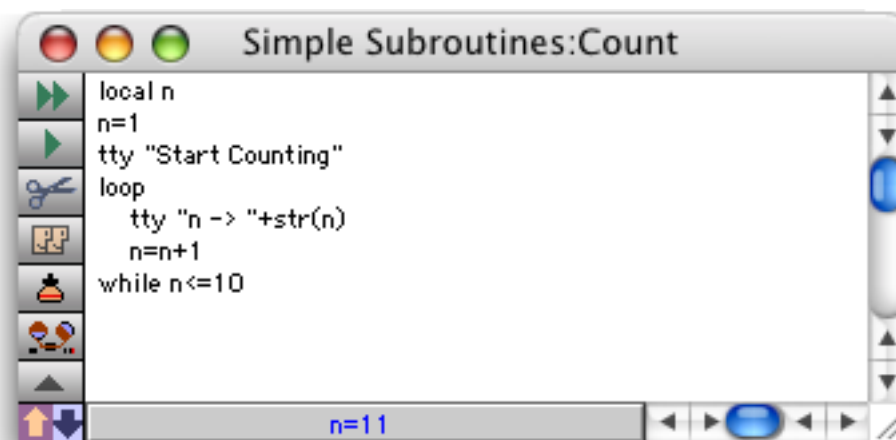
Back in the dark ages of computer history (before 1980) computers generally didn't have fancy debugging systems, and the most common method for finding bugs was inserting "print" statements in the code to type messages on the teletype printer attached to the computer.



By looking at the output of the print statements the programmer could monitor the operation of the program in question. Though we now have many other options for debugging, sometimes simply “printing” can still be the most effective way to monitor program operation. Of course most of us no longer have actual teletypes connected to our computers any more, so Panorama now includes a virtual teletype — the TTY wizard. (Back in the day *TTY* was frequently used as an abbreviation for *teletype*.)



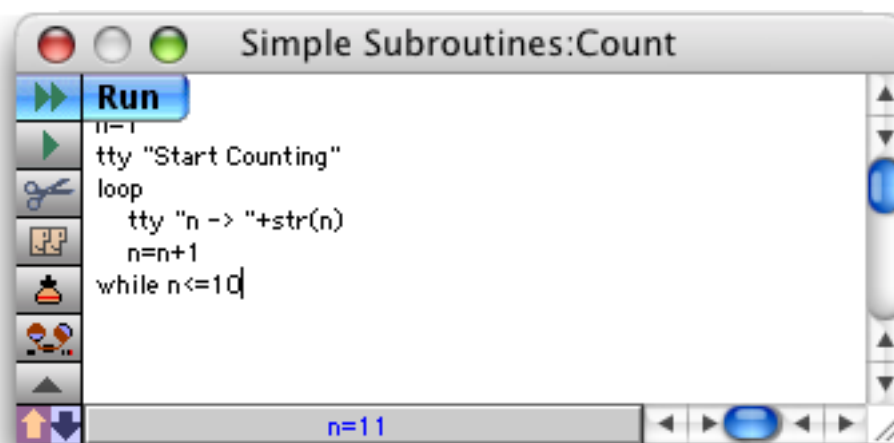
To use this wizard you need to insert one or more `tty` statements in your code. The `tty` statement is kind of like the `message` statement, but instead of displaying an alert it sends the message to the TTY wizard.



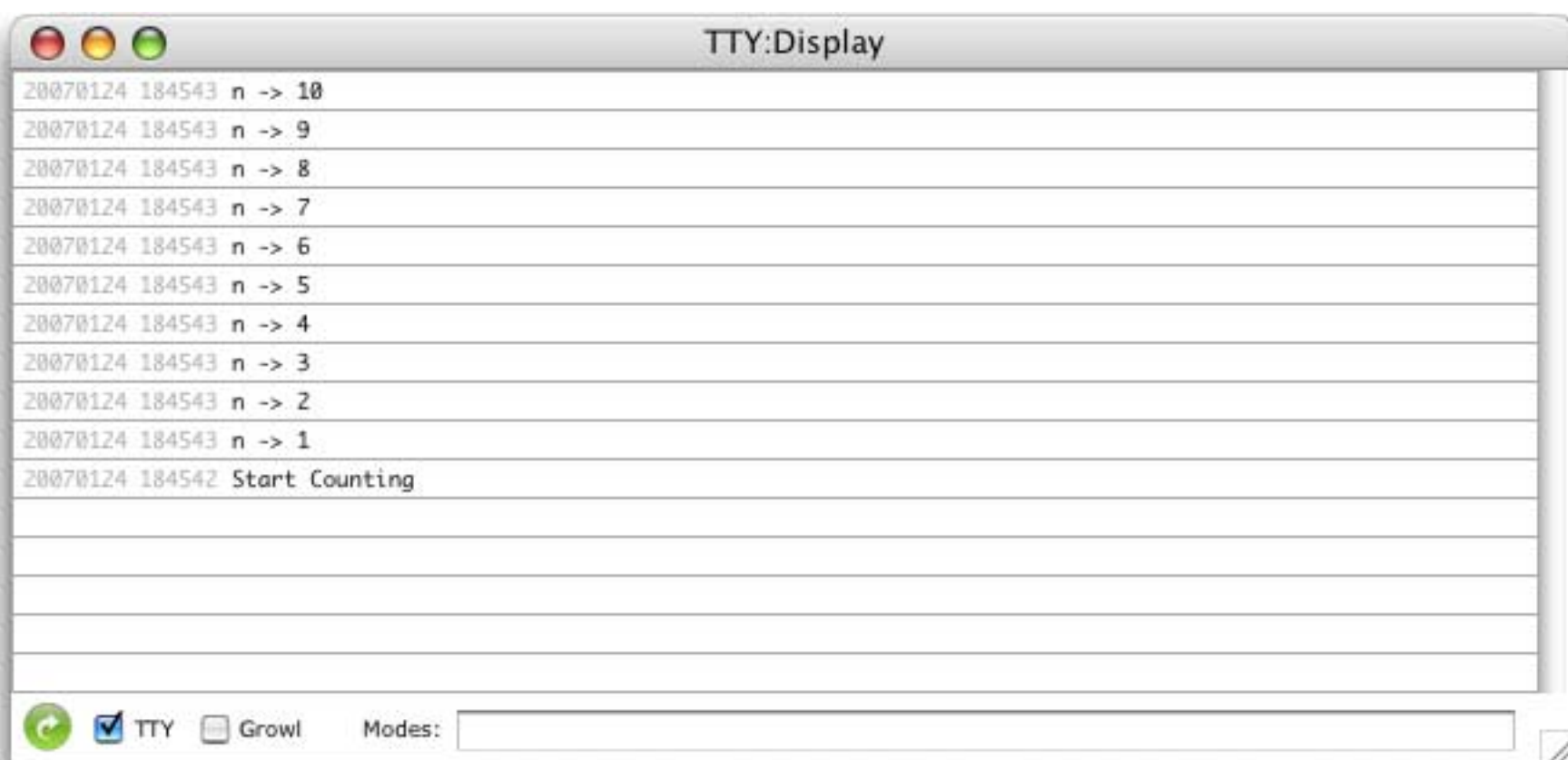
To see the result of the `tty` statements in this procedure you must open the TTY wizard and enable the **TTY** option. (If this option isn’t turned on the `tty` statements will simply be ignored. This means that you can leave the `tty` statements in your procedure permanently if you wish, and only enable them when you want to monitor the output of your program.)



Now go back and run the procedure.

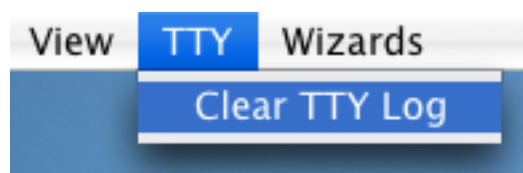


To see the output of the `tty` statements, click on the TTY wizard.



The output is listed in three columns. The first column is the date, in YYYYMMDD format. The second column is the time, in 24 hour HHMMSS format (in the example above the program was run at about 6:45 PM on January 24, 2007). The third column contains the output from the `tty` statements. Notice that the output is listed in reverse order, with the most recent output at the top.

You can keep running the procedure over and over again, or run other procedures with `tty` statements. When the data becomes too unwieldy you can start over by using the **Clear TTY Log** command.



Using TTY with Growl

Growl is a very cool free open source add-on for OS X that displays temporary messages that fade away automatically after a few seconds. In other words, a perfect way to display tty messages! If you don't already have a copy of Growl you can download it from this web site.

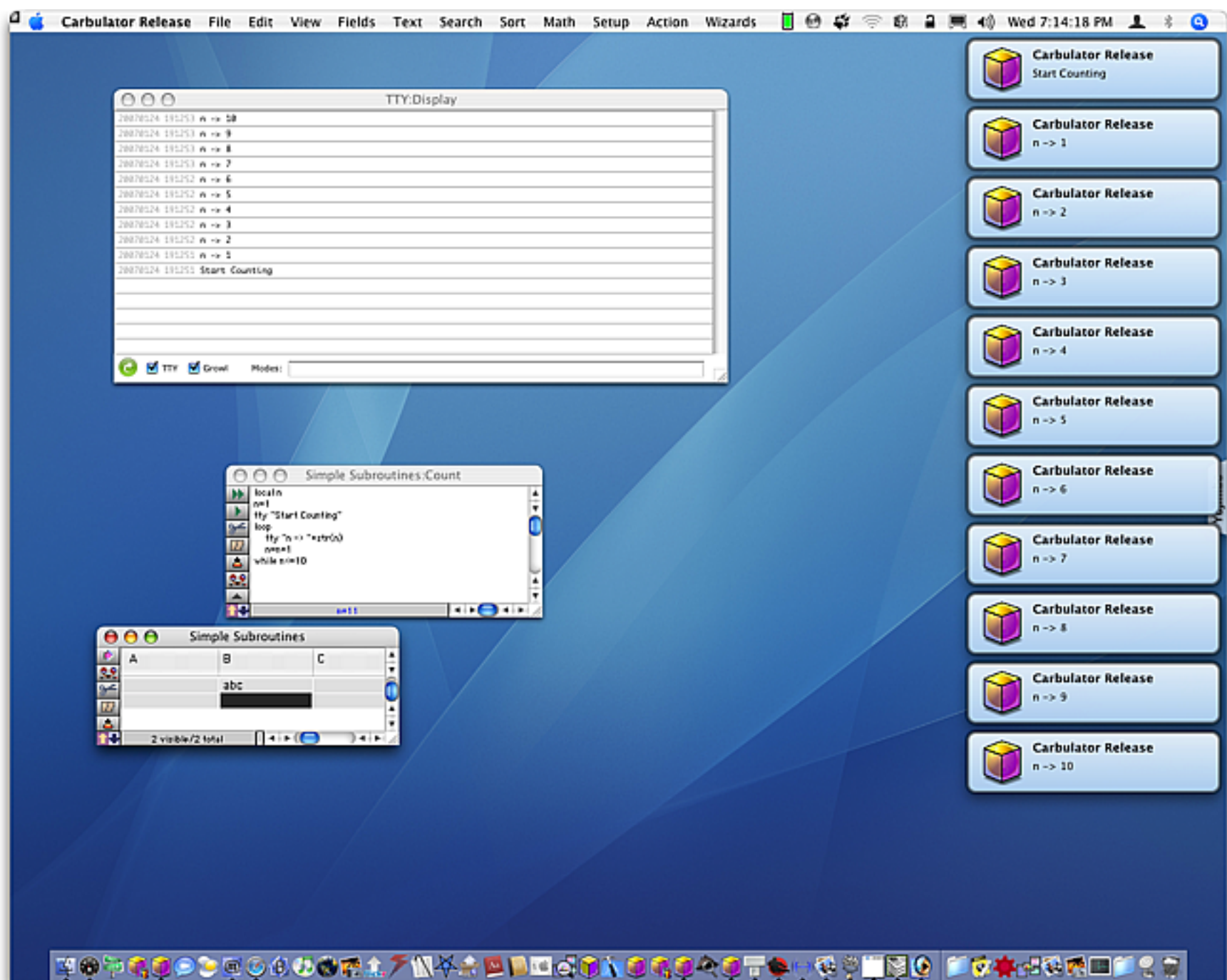
<http://growl.info/>

It's a small download (2 mb) and installs easily. Once it's installed you can enable the Growl option and your tty message can appear in floating "bubbles" that don't stop the program and that fade away after a few seconds.

To use the growl option check the option in the TTY wizard (of course you must install Growl first).



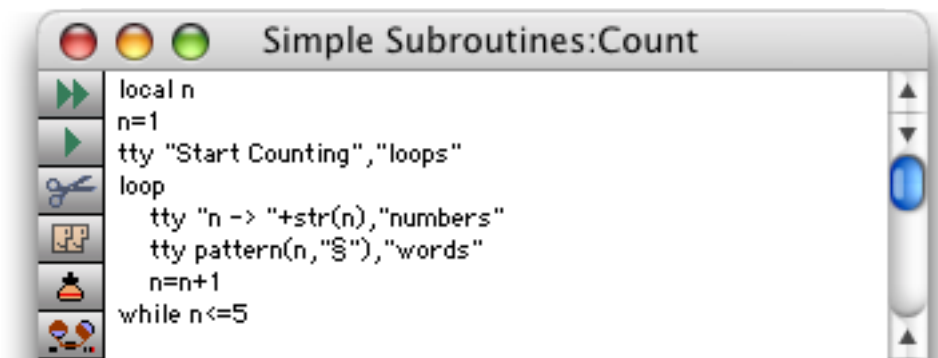
Now when you run the procedure you'll see the growl bubbles appear.



After a few seconds the bubbles will magically disappear (you can also click on them to make them disappear immediately).

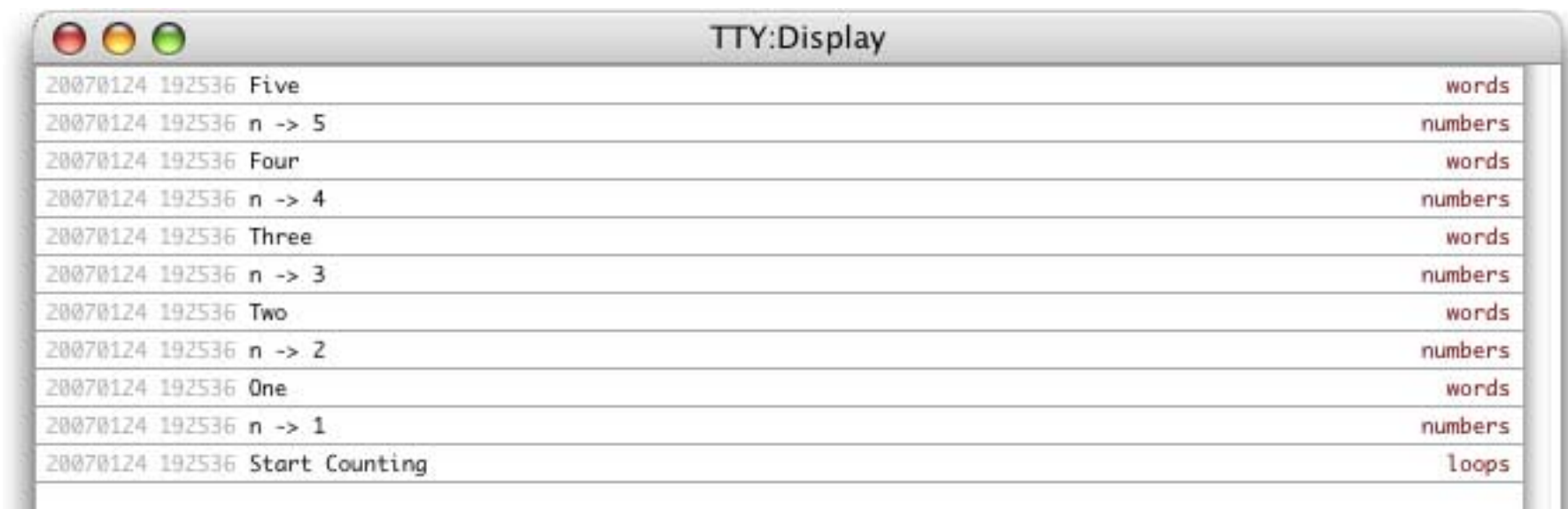
Selective TTY Output (Modes)

The `tty` statement has an optional second parameter which may be used to specify a **mode**. This is simply an identifier that you can use to identify the type or class of information being produced. You can make up any names for modes you like. In this example the procedure uses three modes: `loops`, `numbers` and `words`.



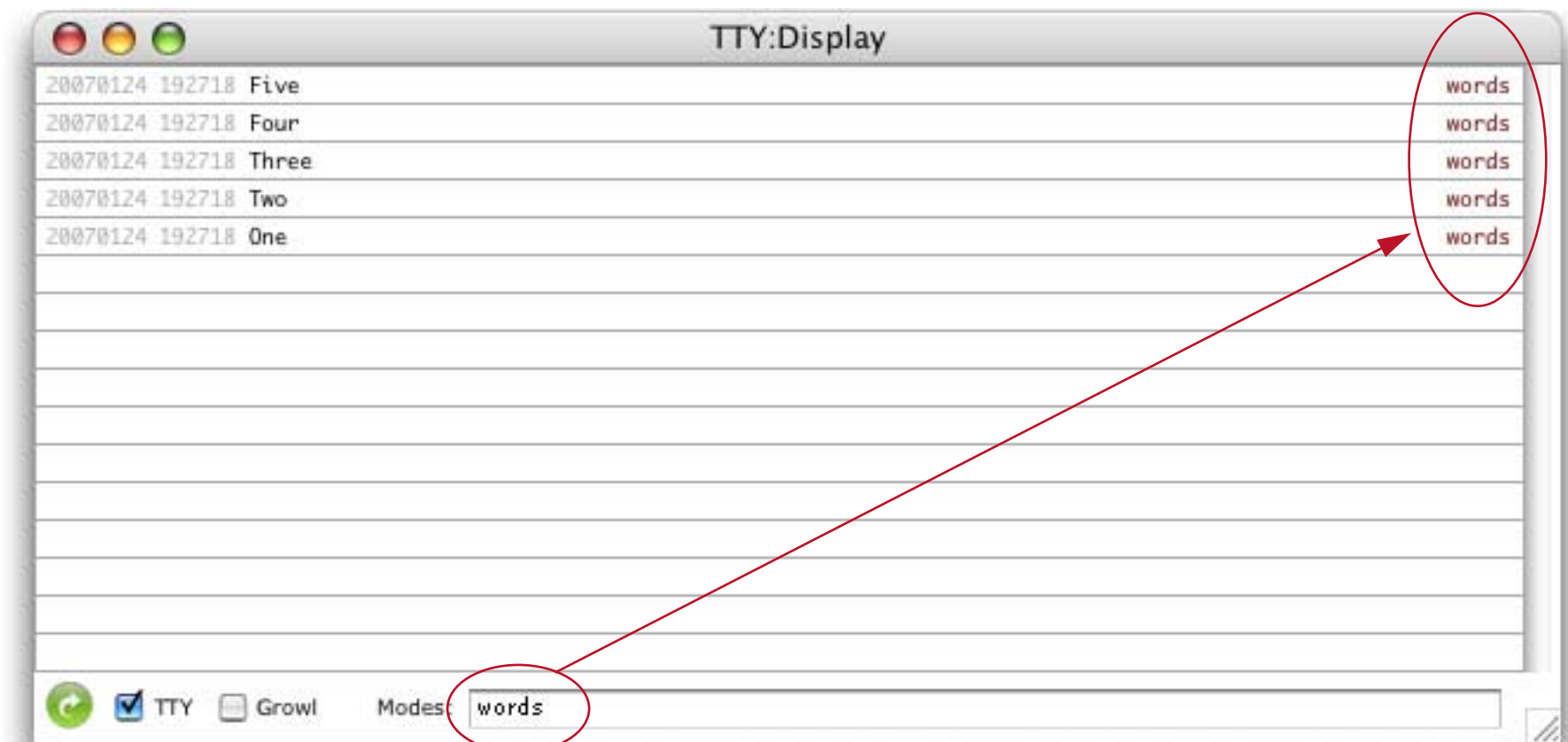
```
local n
n=1
tty "Start Counting","loops"
loop
  tty "n -> "+str(n),"numbers"
  tty pattern(n,"$"),"words"
  n=n+1
while n<=5
```

If you run this procedure you'll see that the mode appears as a fourth column on the right edge of the TTY wizard (I also changed the procedure to only loop 5 times).



Time	Message	Mode
20070124 192536	Five	words
20070124 192536	n -> 5	numbers
20070124 192536	Four	words
20070124 192536	n -> 4	numbers
20070124 192536	Three	words
20070124 192536	n -> 3	numbers
20070124 192536	Two	words
20070124 192536	n -> 2	numbers
20070124 192536	One	words
20070124 192536	n -> 1	numbers
20070124 192536	Start Counting	loops

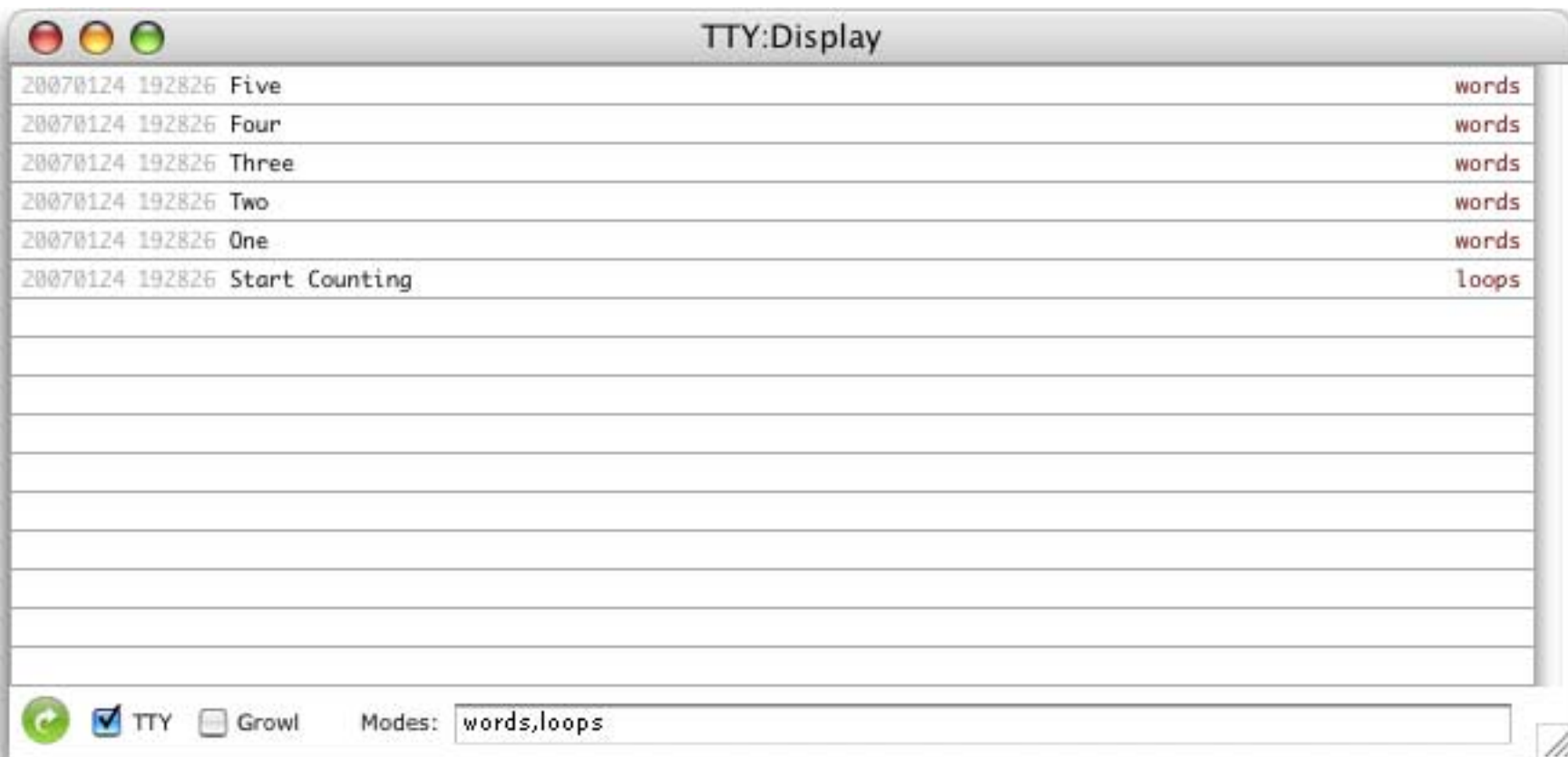
If you type in the name of a mode in the **Modes** box and re-run the program only tty messages with that mode will be displayed.



Time	Message	Mode
20070124 192718	Five	words
20070124 192718	Four	words
20070124 192718	Three	words
20070124 192718	Two	words
20070124 192718	One	words

TTY Growl Modes:

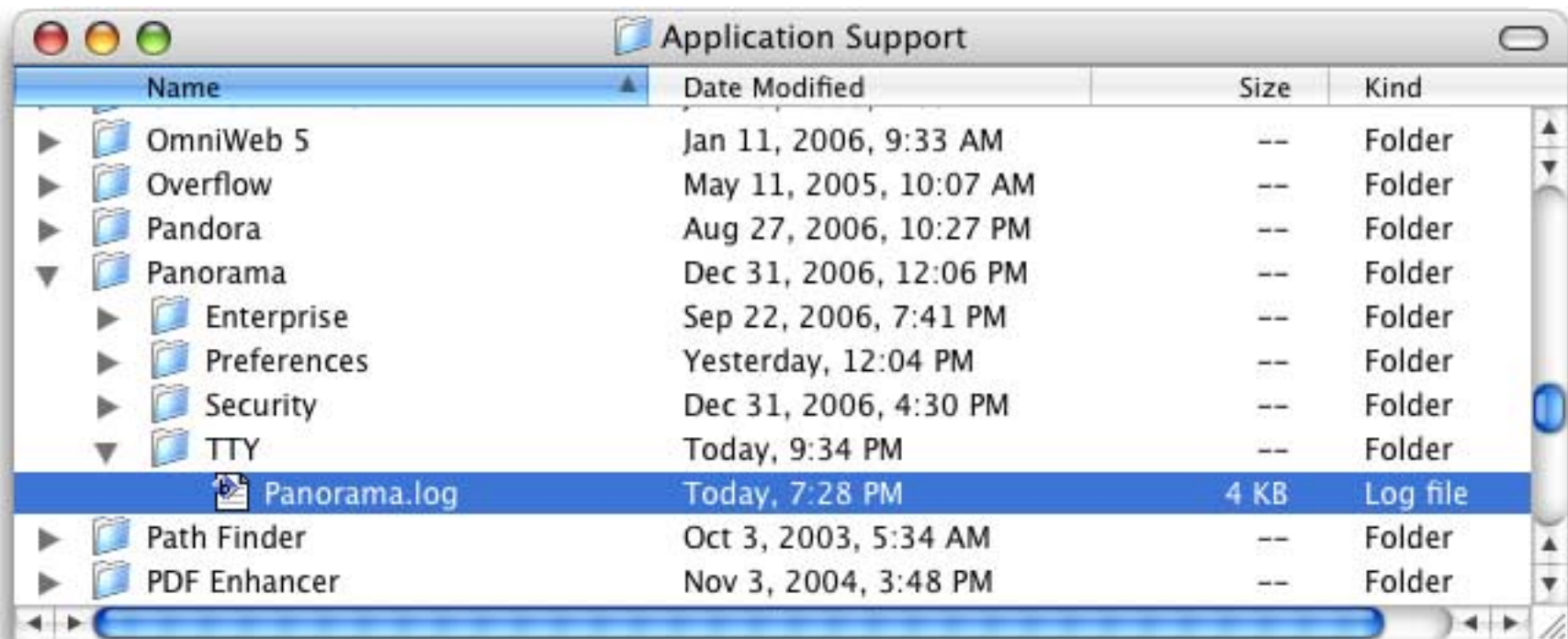
If you type in multiple modes separated by commas, all of the listed modes will be displayed.



Note: Don't type any spaces between the modes, just a comma.

Keeping a Permanent Record

Occasionally you may want to keep a permanent record of the output from the `TTY` statement. The record is kept in a file named `Panorama.log` (or `Panorama Server.log` if you are working with the server) which is found inside the `Application Support` folder, which is inside the `Library` folder in your `Home` folder.



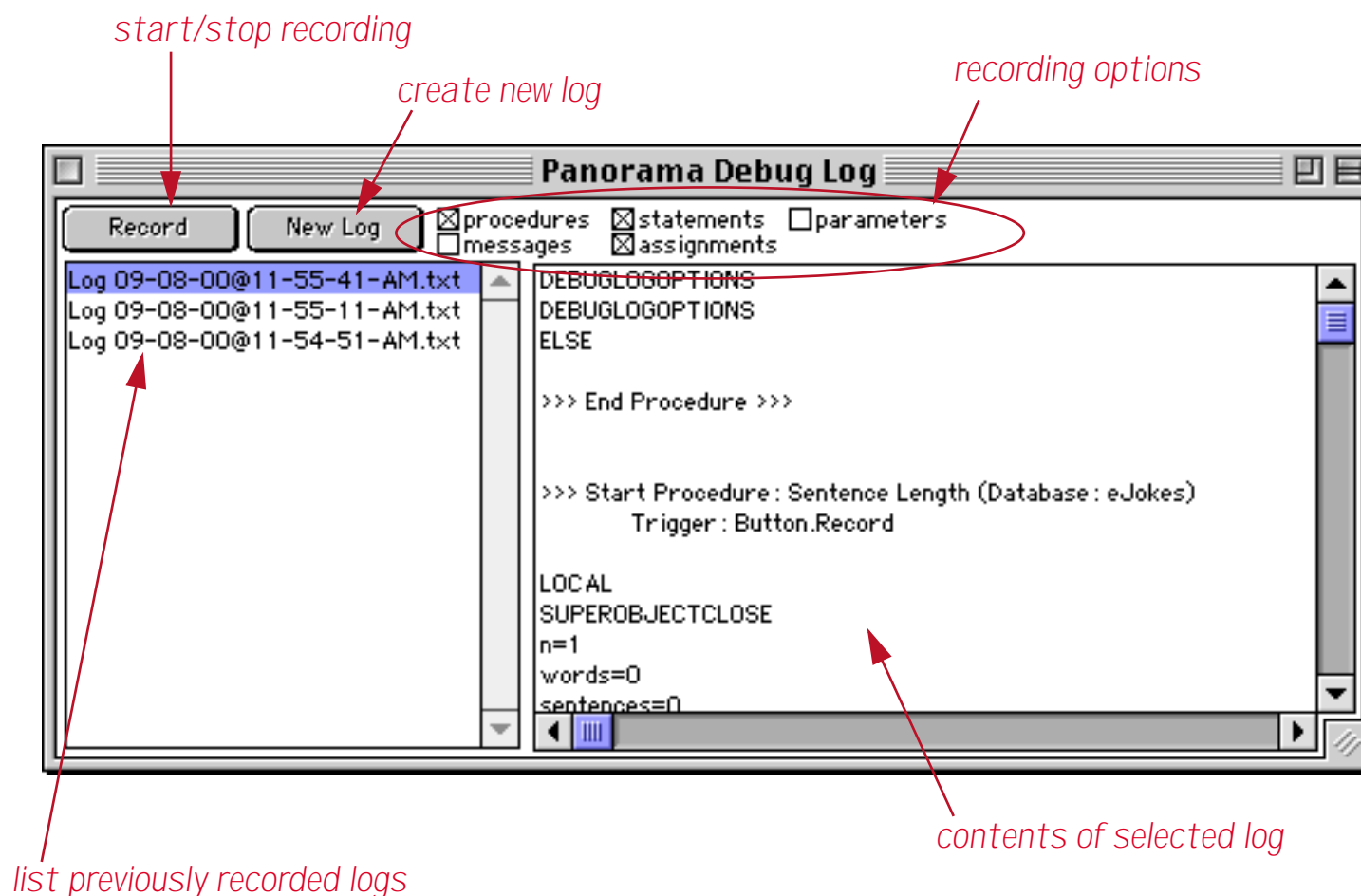
You can move, rename, or duplicate this file to keep a permanent record. The file can be opened with any text editor (BBEdit, TextMate, TextEdit, TextWrangler, etc.) or can be imported into Panorama (the text is tab delimited).

Procedure Debug Log

The procedure debug log was originally developed as an “in house” tool to help debug Panorama itself. It has proved so useful that we have decided to document and make it available for general use. When the debug log is in use Panorama records procedure activity in a text file. Later you can review the text file to trace the actions of your procedure. Although Panorama rarely crashes, when it does the debug log comes in very handy, because it will record the steps taken right up to the crash. This allows you to find out exactly what statement is causing the crash (which explains why this debug log is so useful for our in-house programming of Panorama itself.)

The Procedure Log Window

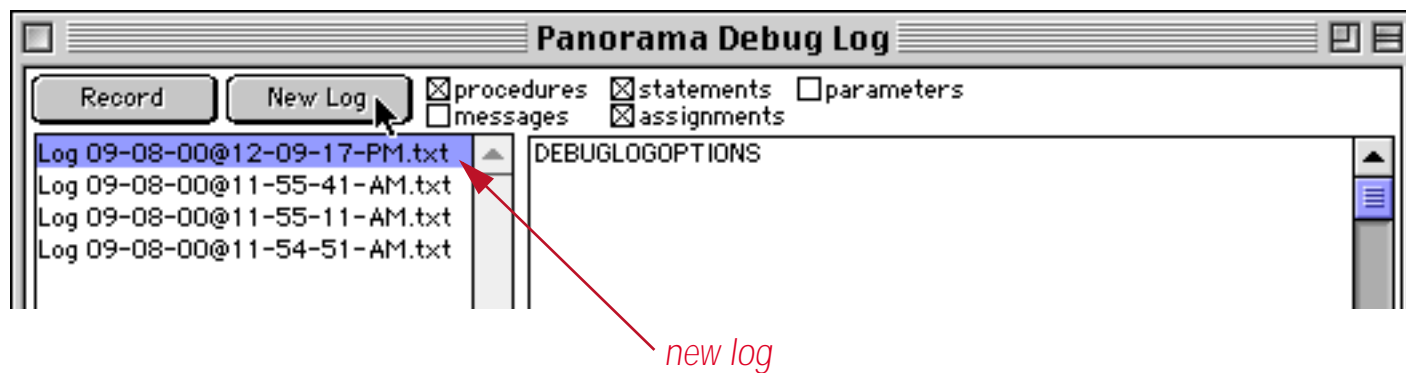
To open the log window choose **Debug Log** from the Wizards menu. When you first open the debug log it looks something like this.



The list on the left hand side of the window shows each of the previously recorded logs. Each log is date and time stamped.

Recording a New Log

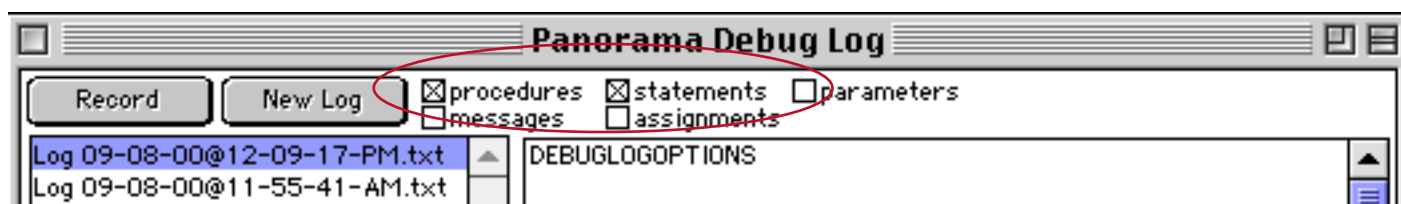
To record a new log, start by pressing the **New Log** button. A new log will be added to the top of the list.



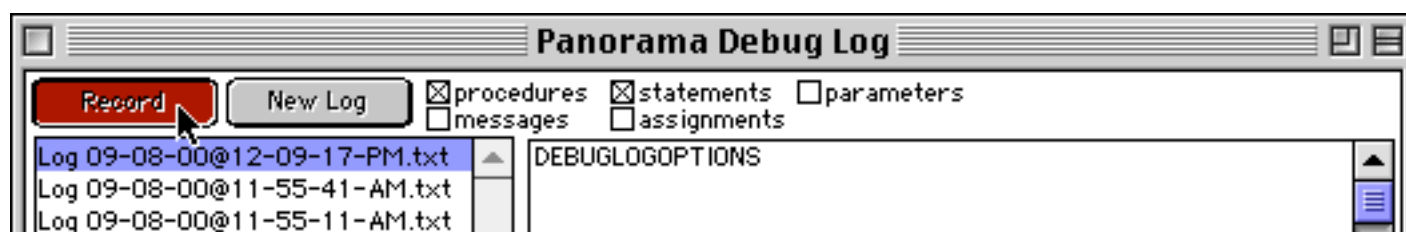
Next, select the recording options for the new log.

Option	Description
procedures	When this option is selected the log will record each time a new procedure starts or finishes, either by being triggered by a menu or button or as a subroutine call. This option can be handy if you are not sure what procedure is triggered by a button. Simply turn on the Debug Log, start recording and press the button. Then check the log to find out which procedure was triggered.
statements	When this option is selected the log will record each statement that is executed. Only the statement itself is recorded, not any parameters (see next section).
parameters	When this option is selected the log will record the values of each statement parameter (see “Decoding Parameters and Assignment Statements” on page 341).
messages	When this option is selected the log will record each logmessage statement (see “The Log-Message Statement” on page 342).
assignments	When this option is selected the log will record each assignment statement (A=B, etc.). See “Decoding Parameters and Assignment Statements” on page 341.

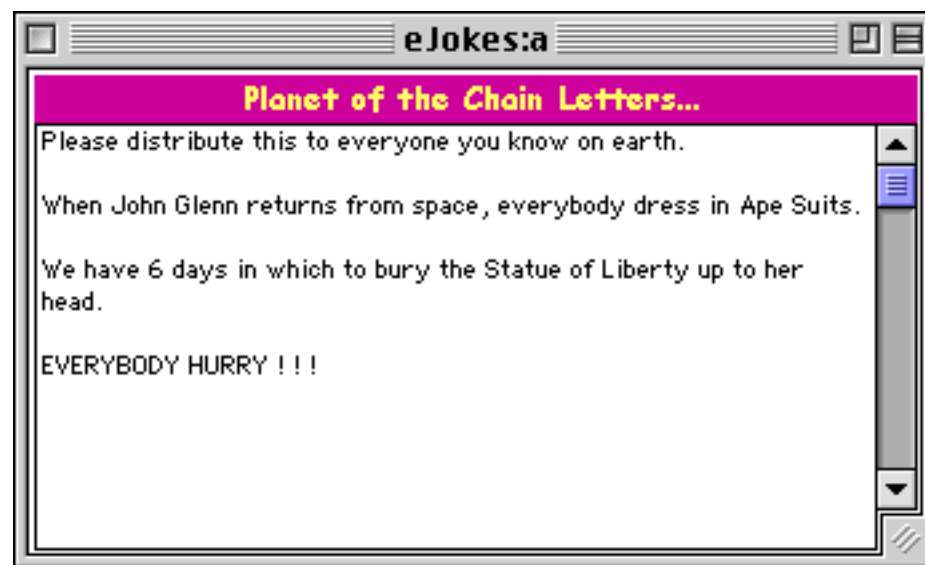
For our first log we'll record only the procedures and statements.



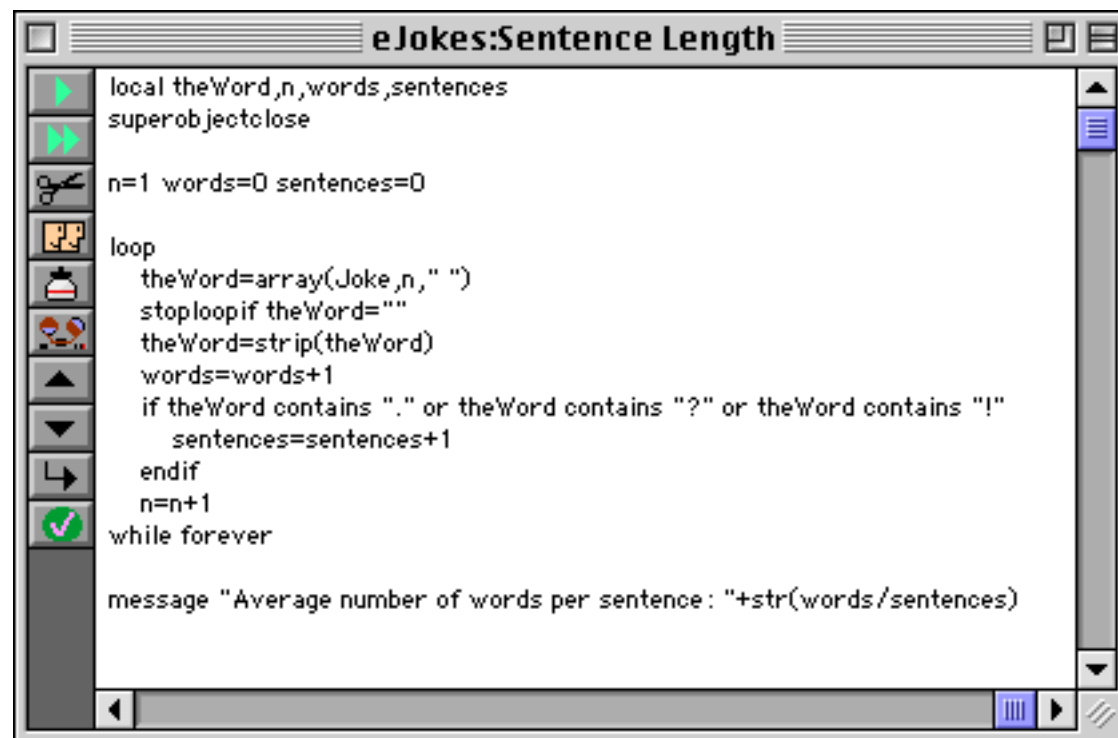
Once the options are set press the **Record** button to start recording. The button will highlight to show that it is recording.



Open the database that contains the procedure you want to test (if it is already open, click on it to bring it to the front).

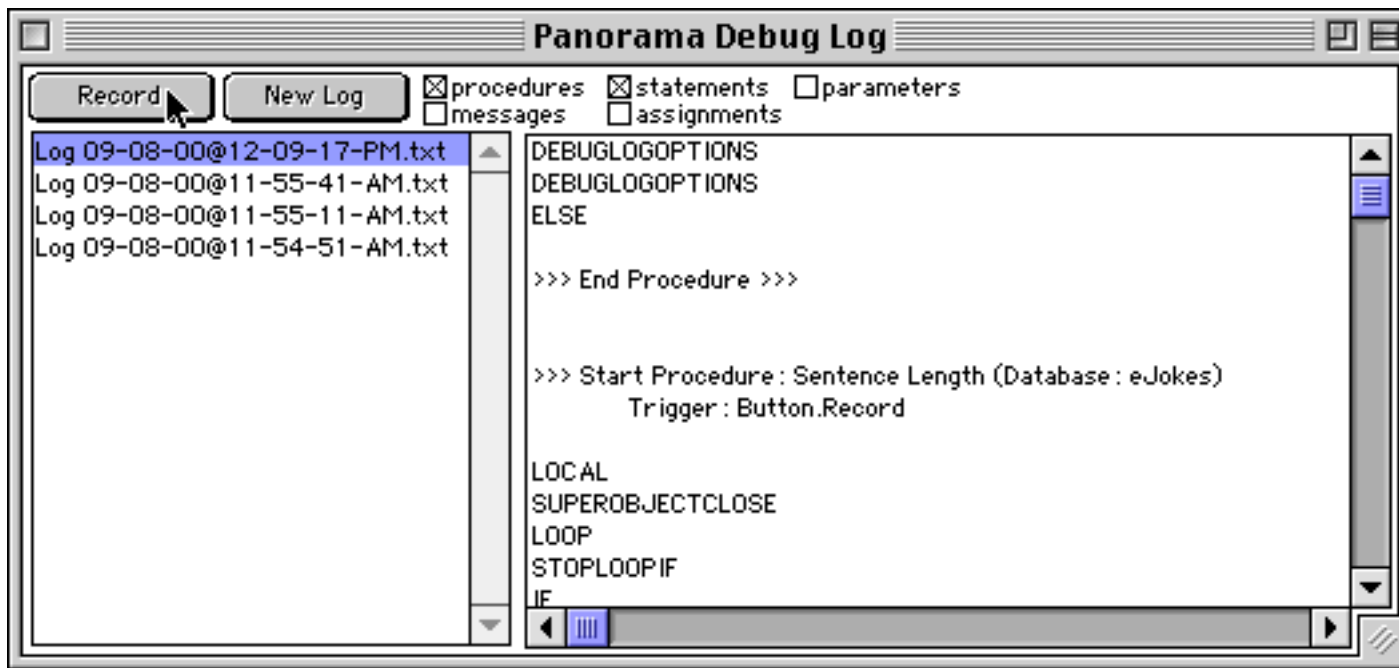


Now perform whatever action it takes to trigger the procedure you want to test — choose the procedure from the **Action** menu, press on a button, enter data, whatever (see “[50 Ways to Trigger a Procedure](#)” on page 355). In this case we are going to test a procedure named **Sentence Length** in the **Action** menu. Here is the text of the procedure.

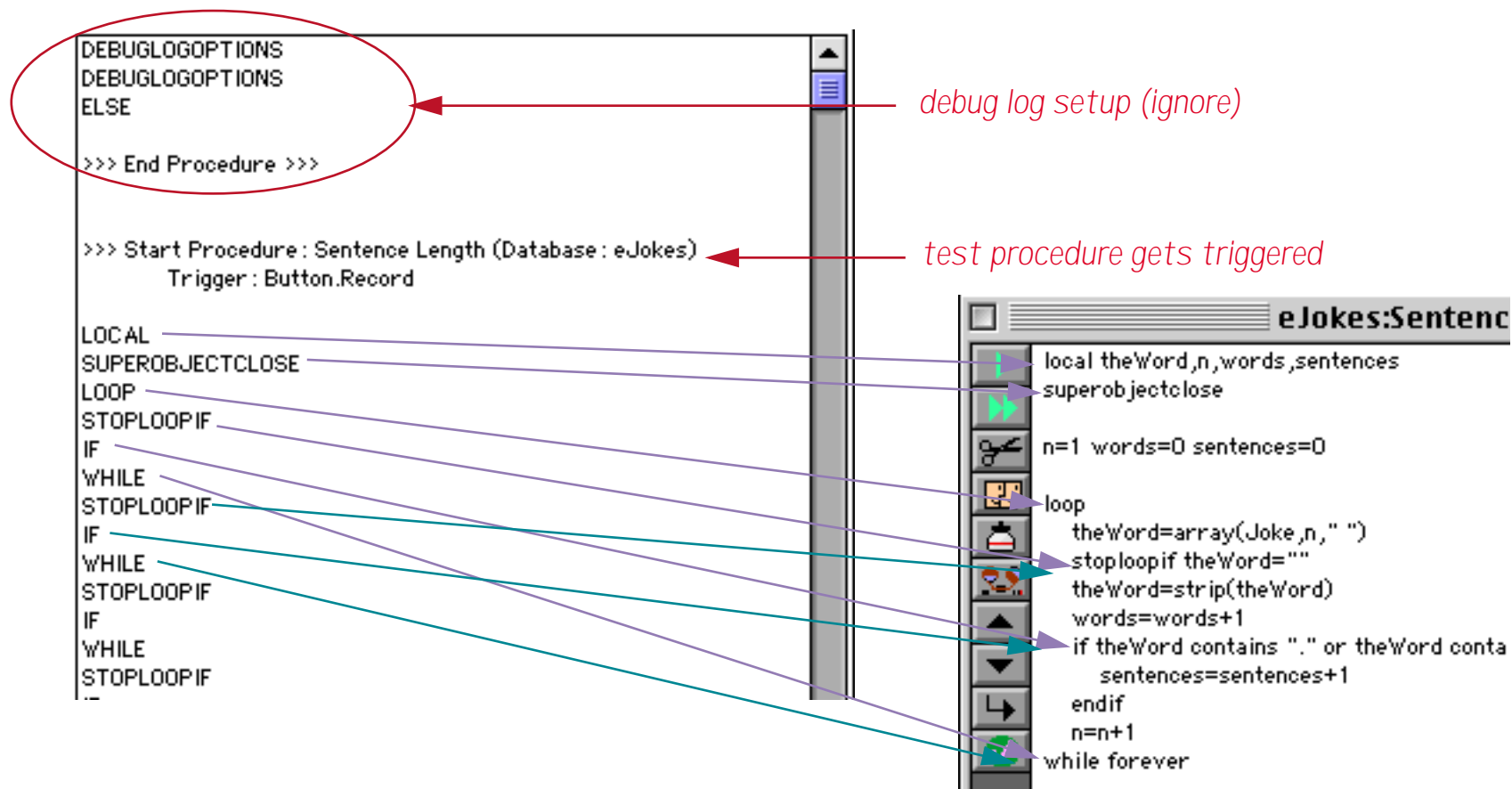


Note: Depending on the recording options you have selected, **the procedure may run much slower than it usually does**. The recording process slows down Panorama’s speed by an order of magnitude or more.

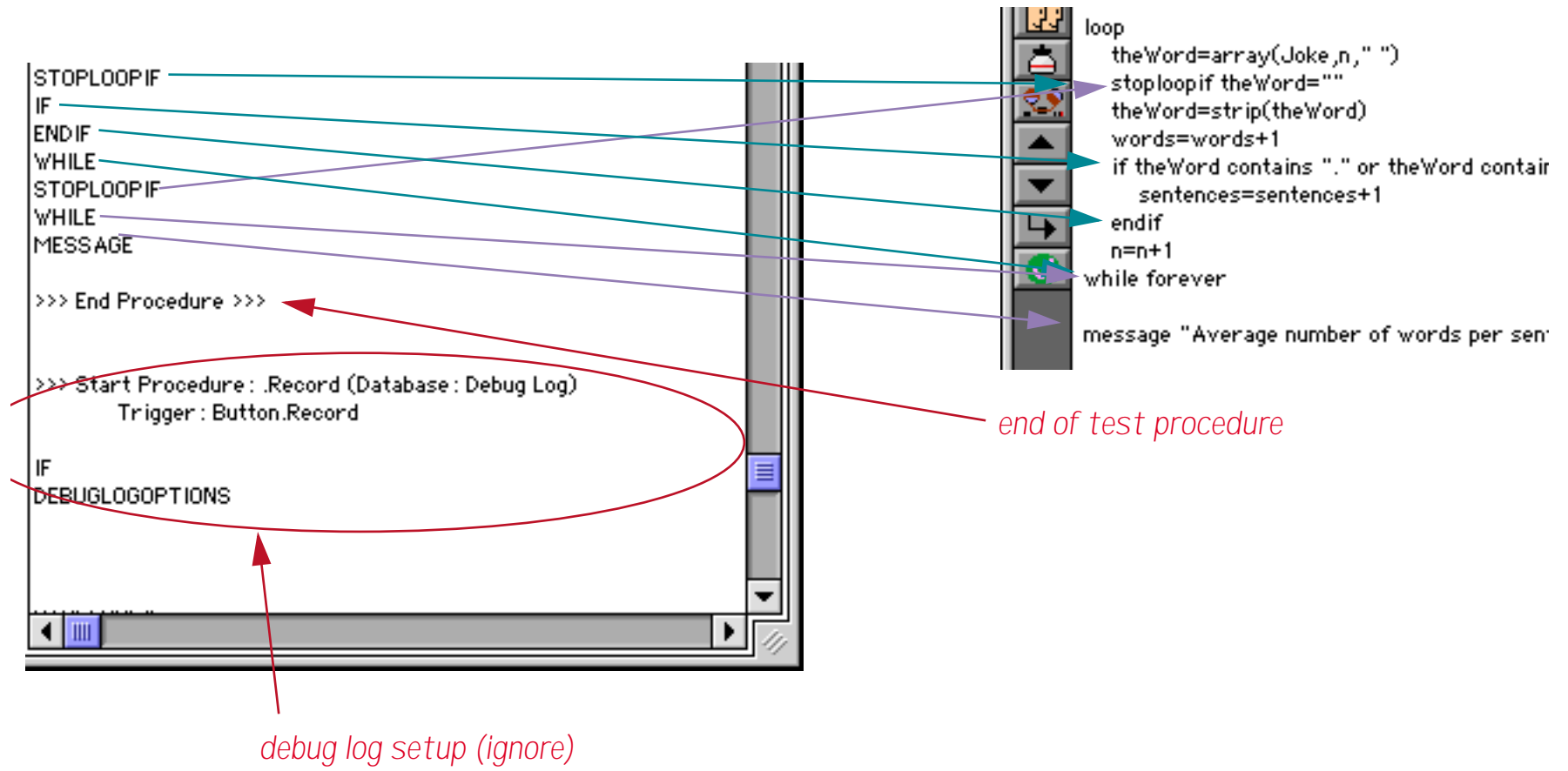
When the procedure has finished running, click on the **Debug Log** window and press the **Record** button. The newly recorded log appears on the right hand side of the window.



At the beginning of the log you may see a few lines caused by the debug log database actually recording itself. You should ignore these lines. Your recording starts with your test procedure being triggered. After that you will see a recording of each statement the procedure performed.



You can continue to trace the steps the procedure took all the way to the end.



Decoding Parameters and Assignment Statements

When the **Parameters** and/or **Assignment** options are enabled the log will contain much more information.

```

>>> Start Procedure : Sentence Length (Database : eJokes)
      Trigger : Button.Record

LOCAL
  Param: theWord,n,words,sentences ← local theWord,n,words,sentences

SUPEROBJECTCLOSE
n=1
words=0
sentences=0
LOOP
n=Please
STOPLOOPIF
  Param: theWord=""
  theWord=Please ← theWord=array(Joke,n," ")
  words=1
  IF
    Param: theWord contains "." or theWord contains "?" or theWord contains "!"
  n=2 ← n=n+1
  WHILE
    Param:
  n=distribute
  STOPLOOPIF
    Param: theWord=""
  theWord=distribute
  words=2 ← words=words+1
  IF
    Param: theWord contains "." or theWord contains "?" or theWord contains "!"
  n=3
  WHILE
    Param:

```

Each statement parameter is logged with the word **Param:** followed by the value of the parameter. Each assignment is logged as the destination (**words=**) followed by the value that the procedure is putting into the destination (**Please**). Notice that in either case the procedure is logging the value and not the formula used to produce the value, for example **n=2**, not **n=n+1**.

The LogMessage Statement

The debug log can quickly generate reams and reams of information that can be tedious to wade through. By inserting the `logmessage` statement in strategic locations you can create a log that shows only the information that is useful to you. Here is a revised version of the procedure with four `logmessage` statements added at strategic spots.

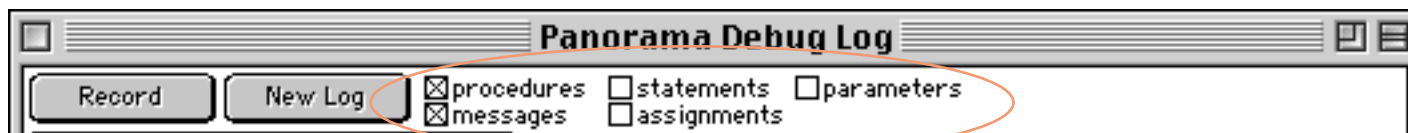
```

local theWord,n,words,sentences
superobjectclose

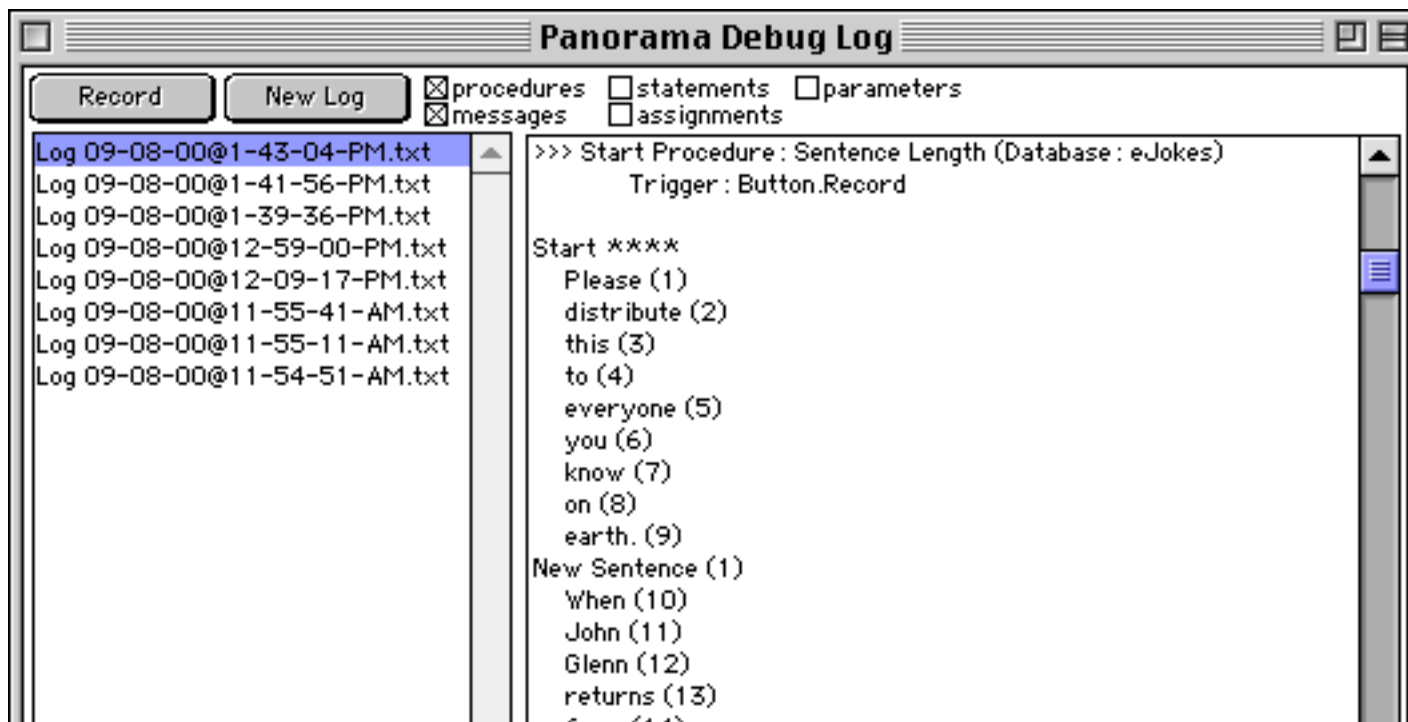
n=1 words=0 sentences=0
logmessage "Start ***"
loop
  theWord=array(replace(replace(Joke,[" "," "],[" "," "]),n," "))
  stoploopif theWord=""
  theWord=strip(theWord)
  words=words+1
  logmessage " "+theWord+" (" +str(words)+")"
  if theWord contains " " or theWord contains "?" or theWord contains "!"
    sentences=sentences+1
    logmessage "New Sentence (" +str(sentences)+")"
  endif
  n=n+1
while forever
  logmessage "Complete ***"

message "Average number of words per sentence: "+str(words/sentences)
    
```

Before recording we'll adjust the log options to only record messages, not statements, parameters or assignments.



The revised log shows only the messages. You can easily see the flow of the procedure as it scans through each word and sentence.



If you look closely at the procedure above (with the `logmessage` statements) you'll notice that the assignment statement at the beginning of the loop is different than in the previous examples.

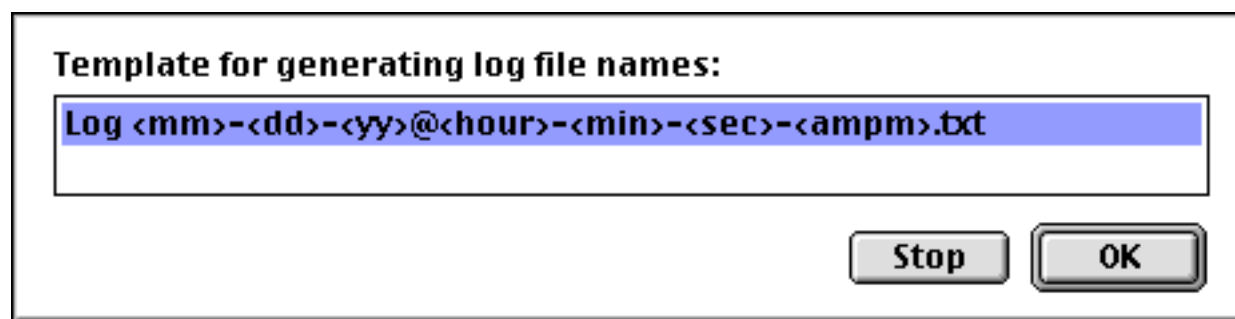
```
theWord=array(replace(replace(Joke,¶," "), " ", " "),n," ")
```

The reason for this change is that in the process of creating the screen shots to demonstrate the `logmessage` statement the log actually showed us that there was a bug in the procedure that caused it to count the number of words incorrectly! The log created with the `logmessage` statements made this bug instantly visible, and hopefully it can do the same for your bugs too!

The Log Menu

When you are finished with a log you can delete it by selecting the log and choosing the **Delete Selected Logs** command from the **Log** menu. To delete every log choose **Delete All Logs** from the **Log** menu.

Panorama normally date and time stamps each log file. You can customize how the log file is created by using the **Edit Log File Template** command from the **Log** menu. This command opens a template that allows you to customize the date/time stamp.



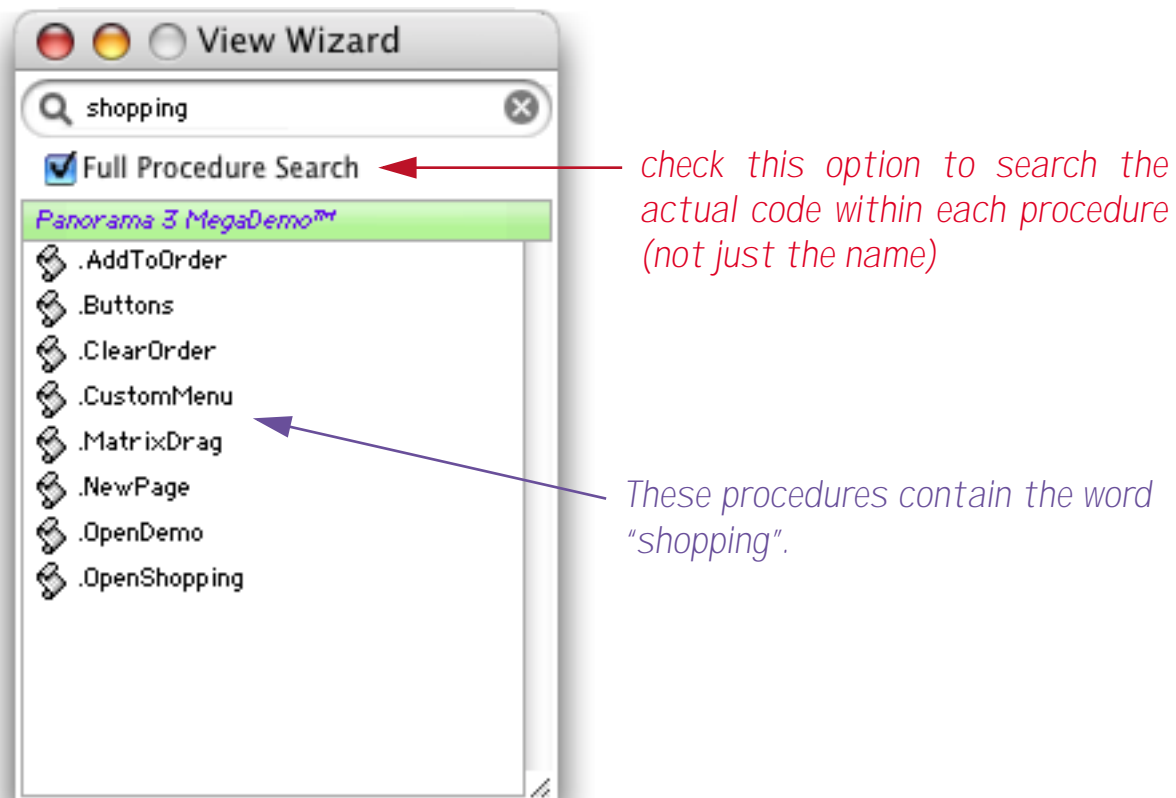
You can customize this template by re-arranging the items.

Using the View Wizard with Procedures

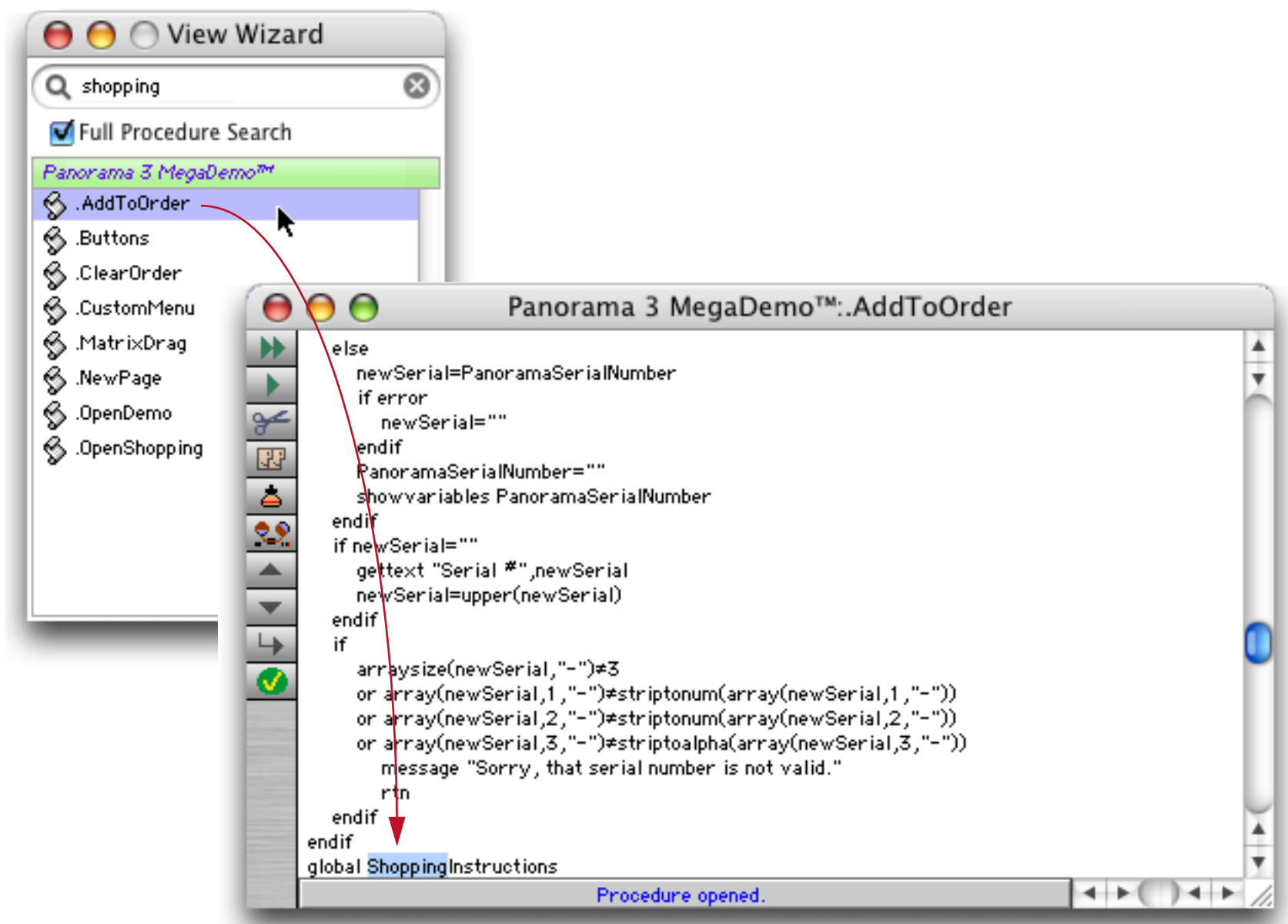
The **View wizard** has some special features for working with procedures. You can search all procedures within a database, list information about procedures, even export procedure source and transfer procedures from one database to another. See “[The View Wizard](#)” on page 173 of the *Panorama Handbook* to learn the basics of working with this wizard.

Searching All Procedures

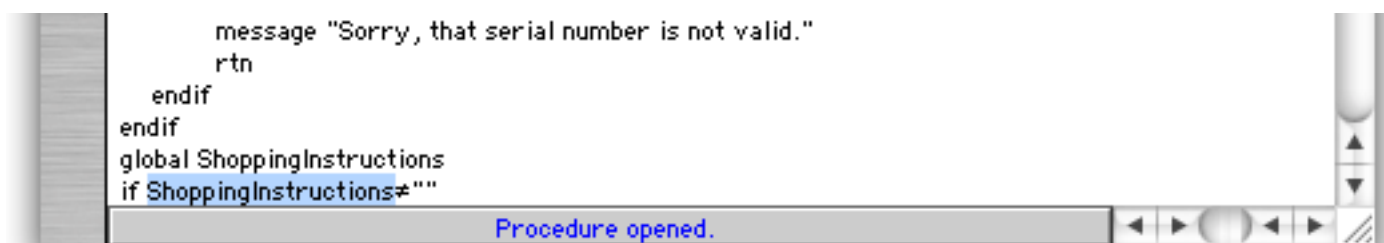
The **View Wizard** has the capability of searching the text of all procedures in a database. Simply check the **Full Procedure Search** option and type in the word or phrase you want to search for. The list will update as you type each key.



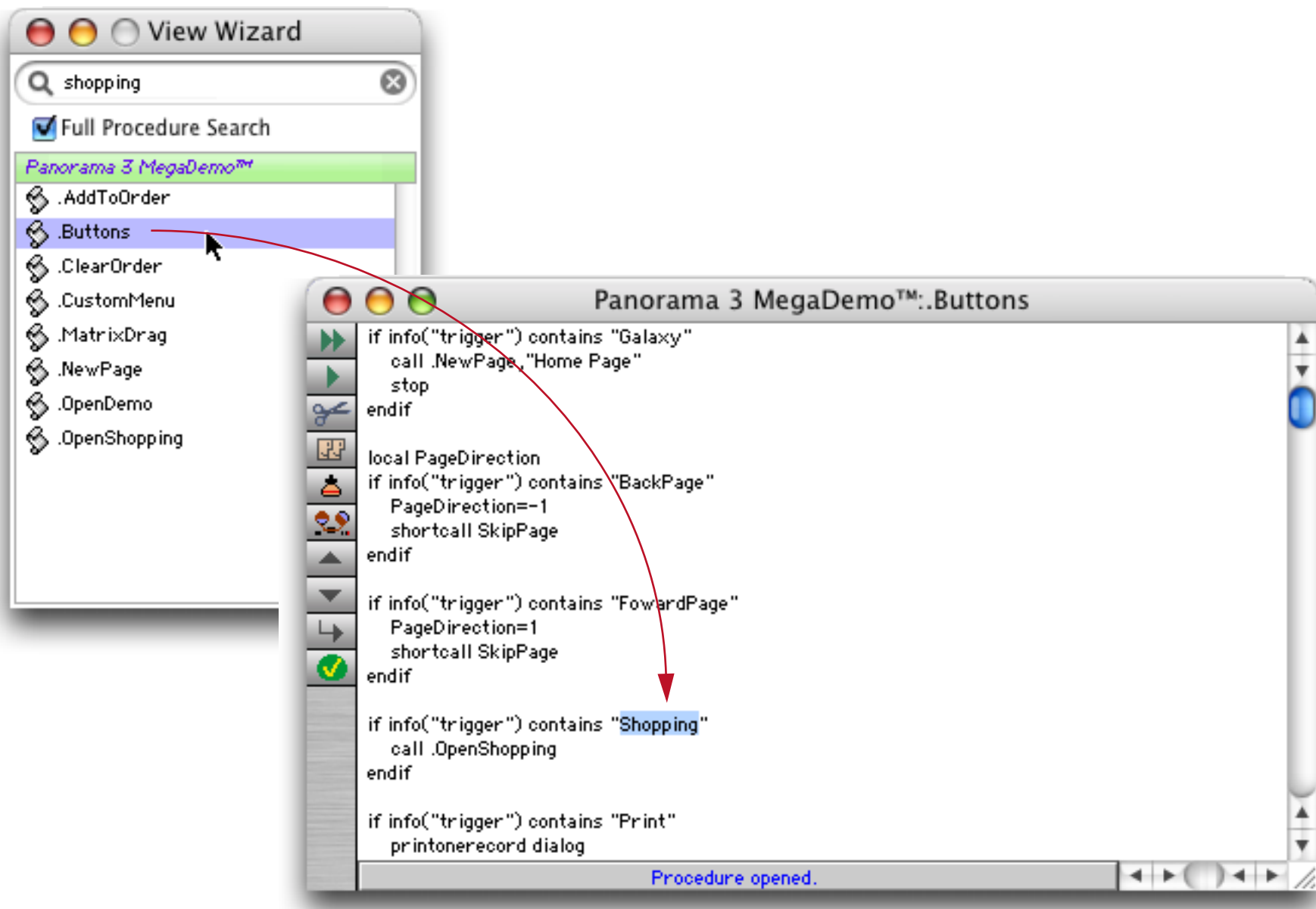
When you double click on one of these procedures the wizard will open the procedure window and automatically locate the first occurrence of the word or phrase.



Choose **Find Next** from the Search menu to find the next occurrence of this word or phrase within the procedure (if any).



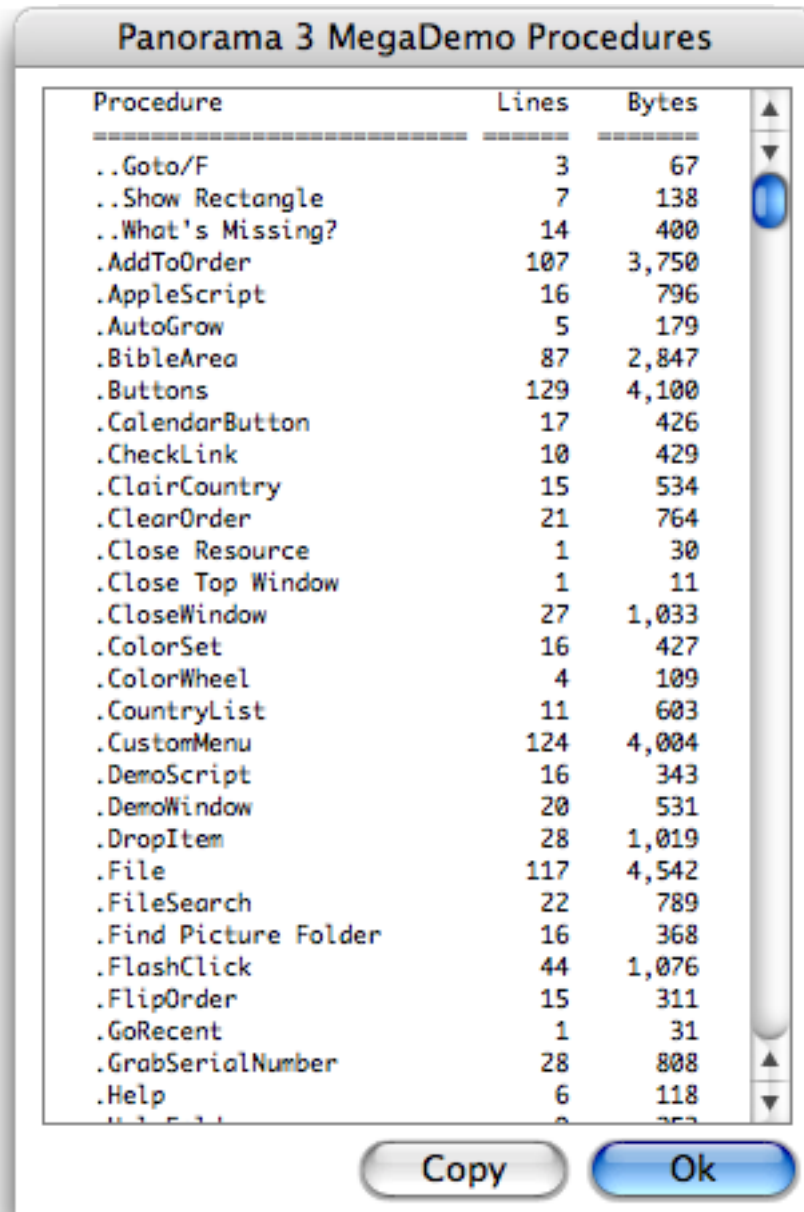
You can repeat using the **Find Next** command until you have located every occurrence of the word or phrase in this procedure. At that point you'll need to go back to the **View Wizard** to continue with the next procedure.



You can continue this process until you have located every occurrence of the word or phrase in the database.

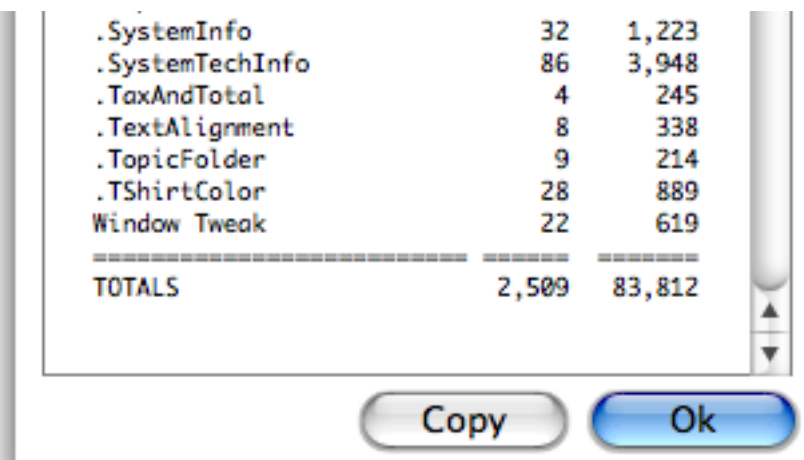
Displaying Source Code Statistics

To see a list of statistics for the procedures in the current database choose **Get Info** from the **Source** menu. A dialog with a list of all procedures in the database will appear, along with the number of lines and bytes in each procedure. (The byte count is the number of characters in the source code, which will be less than the amount shown in the **Memory Usage** window. The **Memory Usage** window shows the total of both the source code and compiled code.)



Procedure	Lines	Bytes
..Goto/F	3	67
..Show Rectangle	7	138
..What's Missing?	14	400
.AddToOrder	107	3,750
.AppleScript	16	796
.AutoGrow	5	179
.BibleArea	87	2,847
.Buttons	129	4,100
.CalendarButton	17	426
.CheckLink	10	429
.ClairCountry	15	534
.ClearOrder	21	764
.Close Resource	1	30
.Close Top Window	1	11
.CloseWindow	27	1,033
.ColorSet	16	427
.ColorWheel	4	109
.CountryList	11	603
.CustomMenu	124	4,004
.DemoScript	16	343
.DemoWindow	20	531
.DropItem	28	1,019
.File	117	4,542
.FileSearch	22	789
.Find Picture Folder	16	368
.FlashClick	44	1,076
.FlipOrder	15	311
.GoRecent	1	31
.GrabSerialNumber	28	808
.Help	6	118

If you scroll down to the bottom you'll see totals for the entire database.

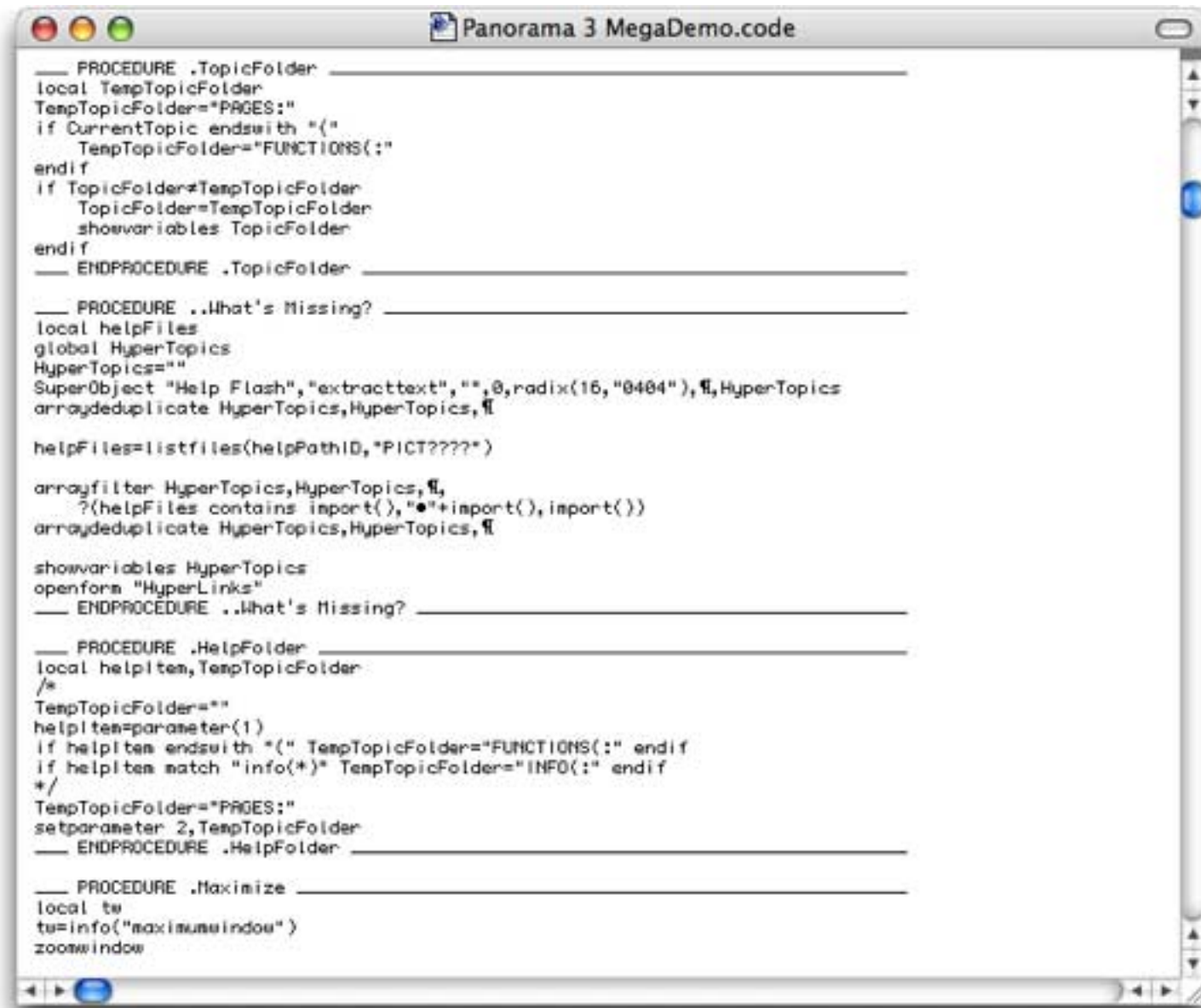


.SystemInfo	32	1,223
.SystemTechInfo	86	3,948
.TaxAndTotal	4	245
.TextAlignment	8	338
.TopicFolder	9	214
.TShirtColor	28	889
Window Tweak	22	619
=====		
TOTALS	2,509	83,812

You can also copy this information to the clipboard using the **Copy** button.

Exporting and Importing Procedure Source Code

To export all of the procedures in a database into a text file use the **Export Source** command in the **Source** menu. The exported file will look something like this:



```

____ PROCEDURE .TopicFolder _____
local TempTopicFolder
TempTopicFolder="PAGES:"
if CurrentTopic endswith "("
  TempTopicFolder="FUNCTIONS(:"
endif
if TopicFolder≠TempTopicFolder
  TopicFolder=TempTopicFolder
  showvariables TopicFolder
endif
____ ENDPROCEDURE .TopicFolder _____

____ PROCEDURE ..What's Missing? _____
local helpFiles
global HyperTopics
HyperTopics=""
SuperObject "Help Flash","extracttext","",0,radiX(16,"0404"),f,HyperTopics
arraydeduplicate HyperTopics,HyperTopics,f

helpFiles=listfiles(helpPathID,"PICT????")

arrayfilter HyperTopics,HyperTopics,f,
  ?(helpFiles contains import(),"•"+import(),import())
arraydeduplicate HyperTopics,HyperTopics,f

showvariables HyperTopics
openform "HyperLinks"
____ ENDPROCEDURE ..What's Missing? _____

____ PROCEDURE .HelpFolder _____
local helpItem,TempTopicFolder
/*
TempTopicFolder=""
helpItem=parameter(1)
if helpItem endswith "(" TempTopicFolder="FUNCTIONS(:" endif
if helpItem match "info(*)" TempTopicFolder="INFO(:" endif
*/
TempTopicFolder="PAGES:"
setparameter 2,TempTopicFolder
____ ENDPROCEDURE .HelpFolder _____

____ PROCEDURE .Maximize _____
local tw
tw=info("maximussindow")
zoomwindow

```

You can open this file in any text editor to view or modify it. However if you are going to re-import it (see below) be sure that you don't disturb the PROCEDURE and ENDPROCEDURE lines.

To import a text file of previously exported procedures use the **Import Source** command in the **Source** menu. You can import the text file into a different database or back into the original database (presumably you have modified the procedures before doing this).

If you want to transfer one or more procedures from one database to another first export all of the procedures to a text file, then use a text editor to remove the procedures that you don't want to transfer. Then import the modified text file into the second database.

Cross Referencing

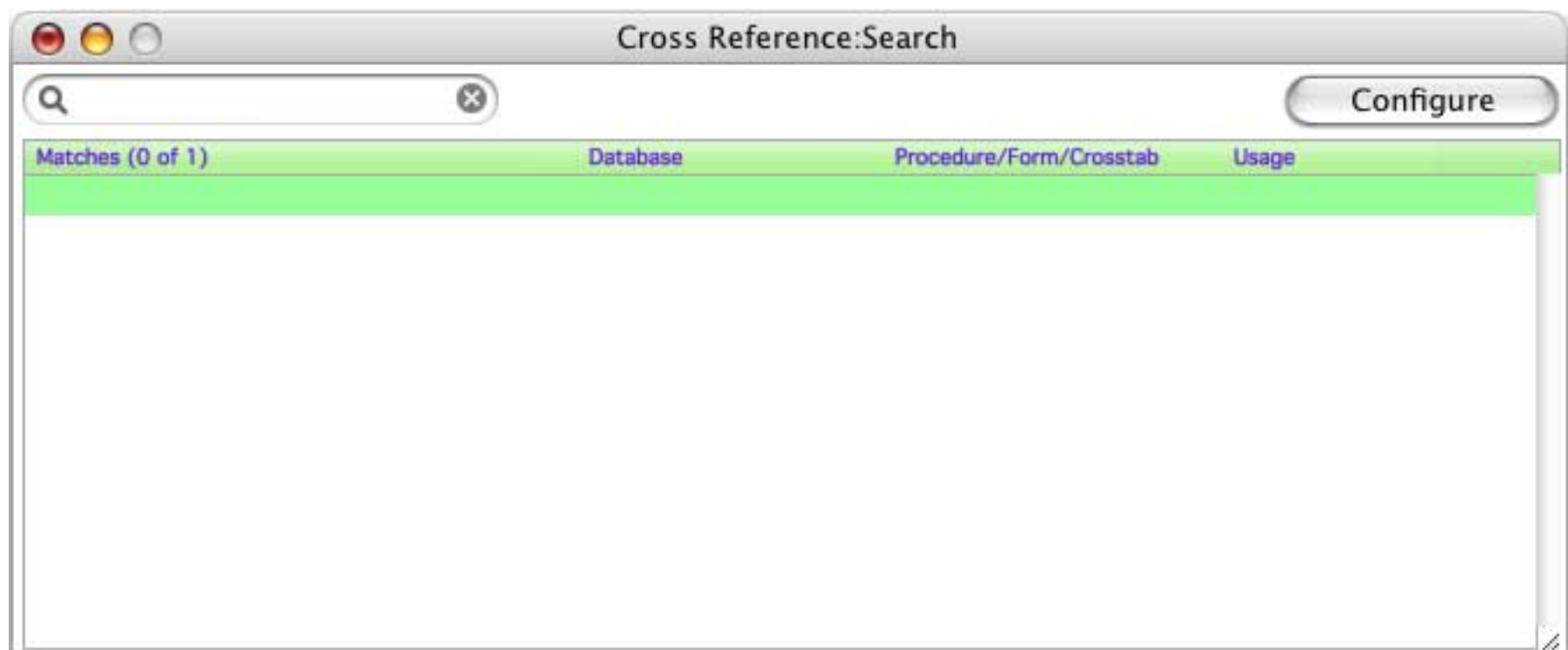
A complex real world system (accounting, reservations, order entry, etc.) created with Panorama may involve a dozen files with hundreds of fields, variables, procedures, forms, etc. Keeping track of all this information in your head can be a monumental task.

Panorama's **Cross Reference** database feature can help make this task manageable. A cross reference database keeps track of all the items in one or more databases: every field, every variable, every procedure, every form—every everything. Not only does the cross reference database keep track of where these items are defined, but also everywhere they are used. For example, suppose your database has a field named **Title**. A cross reference database can tell you that this field is used in the **Entry**, **List**, and **Label** forms, and is also used in the procedures **.NewRecord** and **Search**. Or you could use a cross reference database to find out that the **.LastYear** procedure is triggered by buttons in the **Entry** and **Annual Report** forms. As your database applications become more complicated you'll find that a cross reference database is an invaluable tool to help you sift through a mountain of databases and programming.

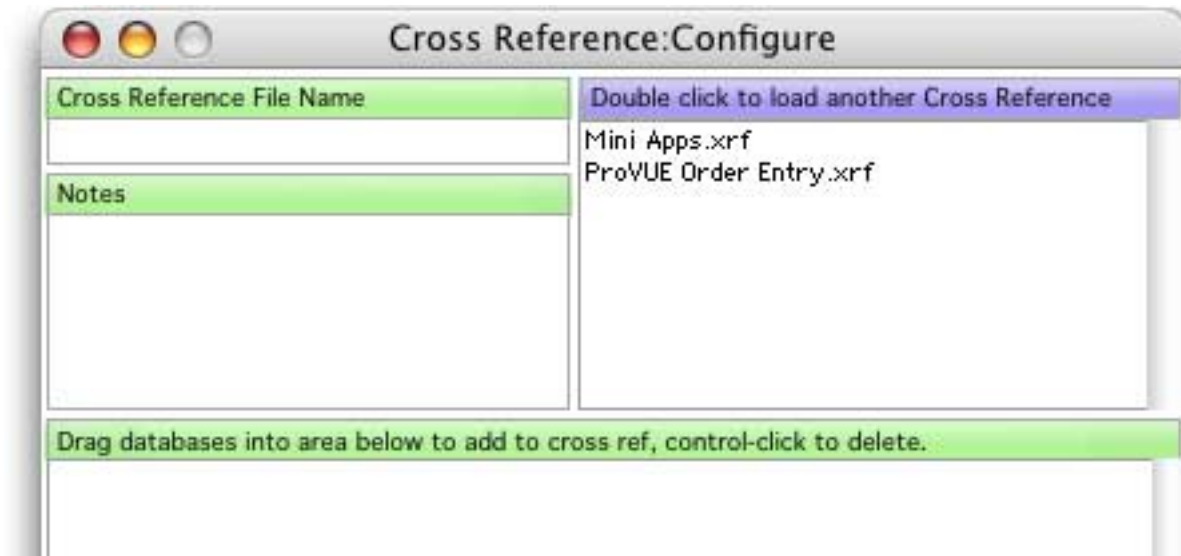
Note: You can also use the **View Wizard** to search through procedures (see "[Searching All Procedures](#)" on page 178). However unlike a **Cross Reference** database the **View Wizard** cannot search through forms, crosstabs or the design sheet.

The Cross Reference Wizard

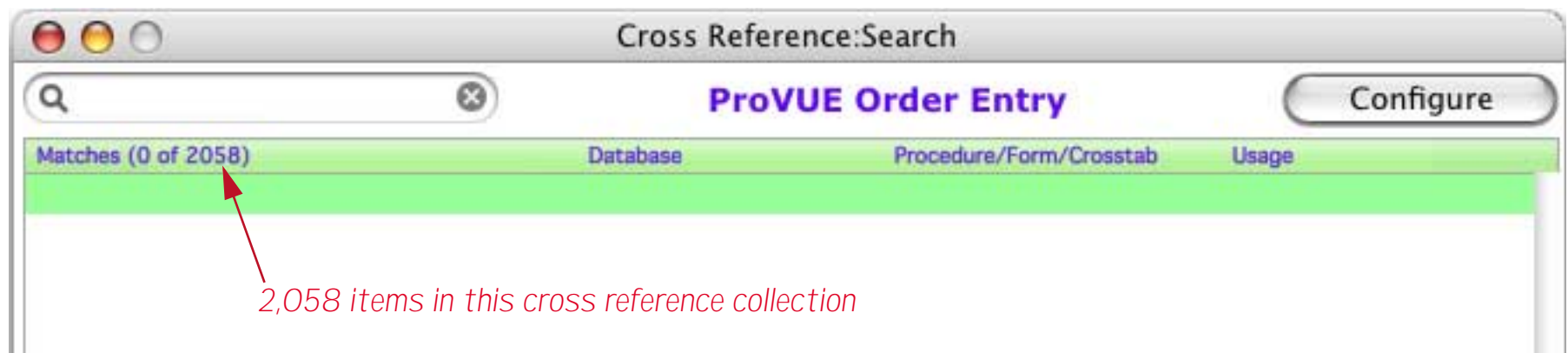
You'll find the **Cross Reference** wizard in the Developer Tools submenu of the Wizards menu.



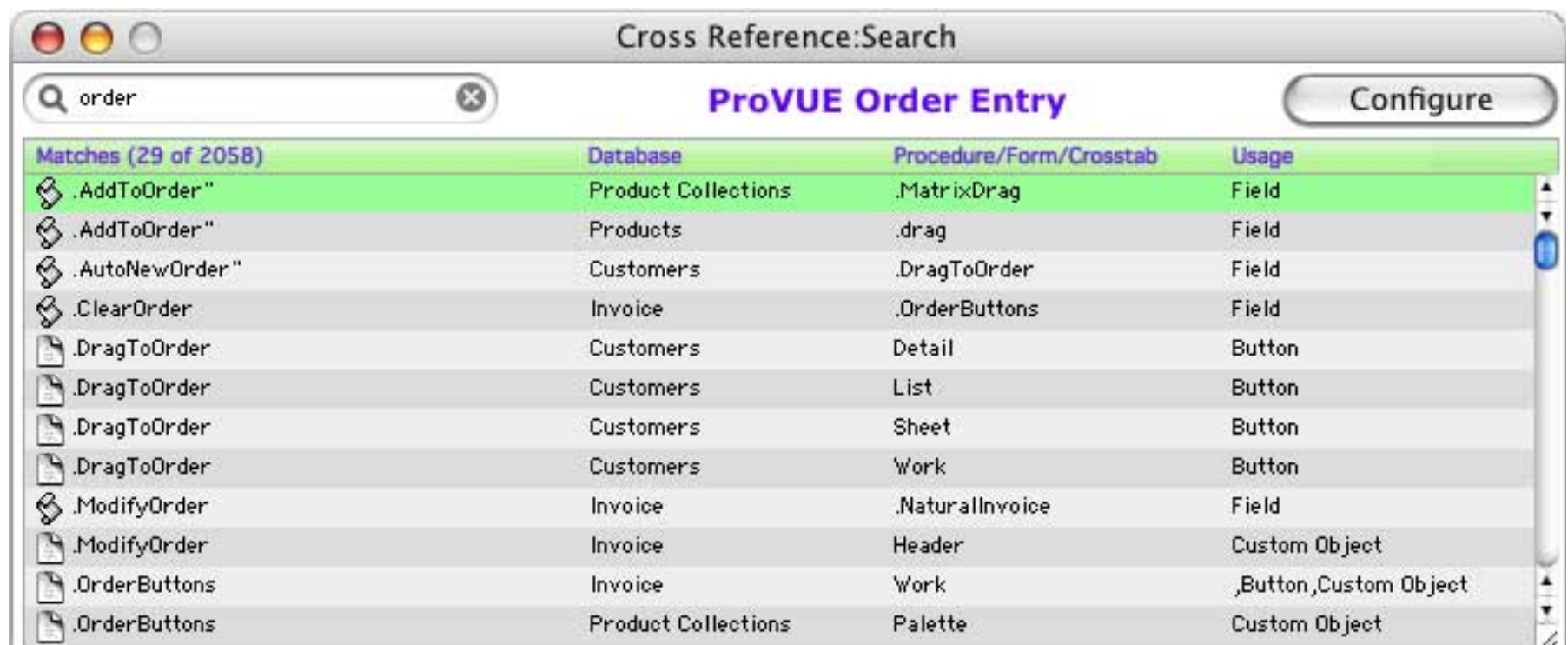
Before you can use this wizard you'll need to configure it, so press the **Configure** button.







For now we'll simply use one of the previously saved cross references that comes with Panorama. Double click on **ProVUE Order Entry.xrf**. The previously configured cross reference appears in the main window. This cross reference contains a combined index for six databases.



This cross reference contains a combined index for six databases. As you start typing into the search box in the upper left, the wizard shows you where that text appears in any of the forms, crosstabs, in the six databases. There are 29 fields, forms or procedures that contain the phrase "order."



The wizard display contains four columns. The leftmost column contains the full contents of the text that matched your search, along with an icon that identifies the type of view.

	Field (data sheet)
	Procedure (script)
	Form
	Crosstab

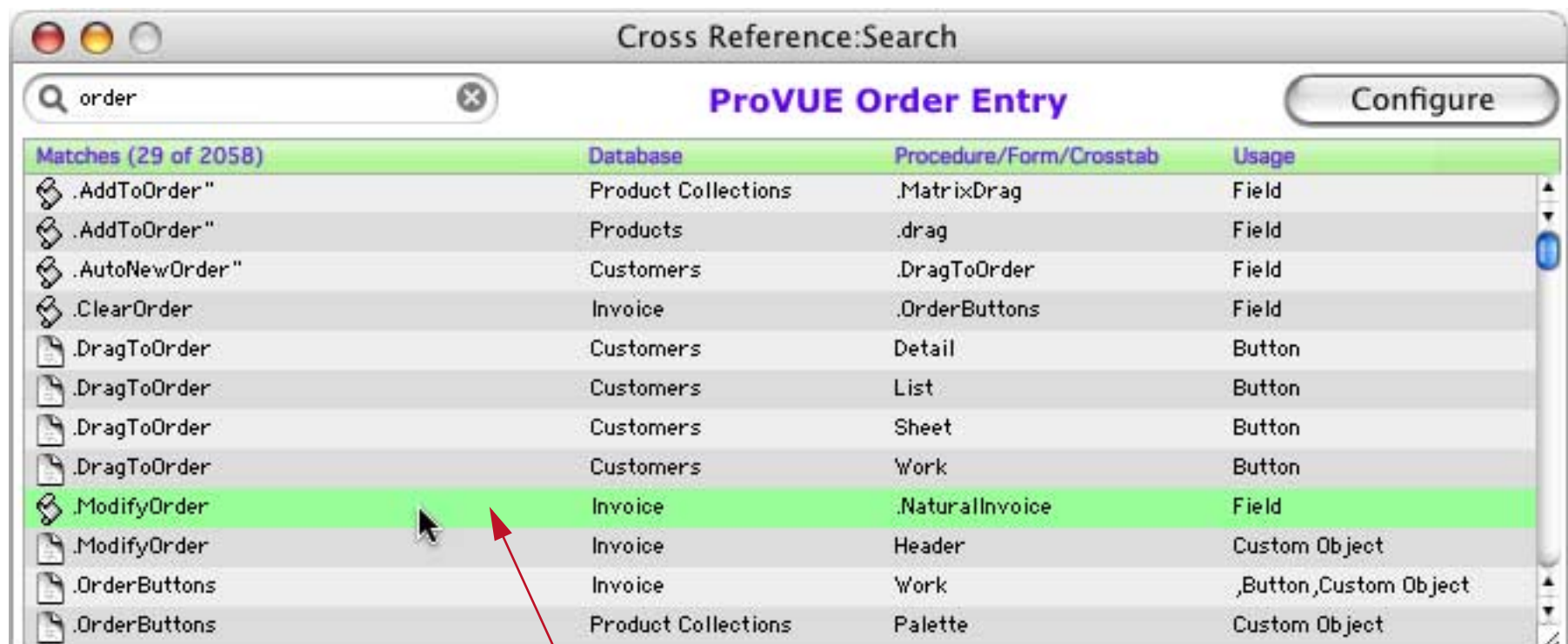
The **Database** column contains the name of the database that contains this item.

The **Procedure/Form/Crosstab** column contains the name of the procedure, form or crosstab that contains this item.

The **Usage** column tells how the matched item is used in this instance.

Opening a Form, Procedure or Crosstab

To open any of the items in the list, just double click on it.



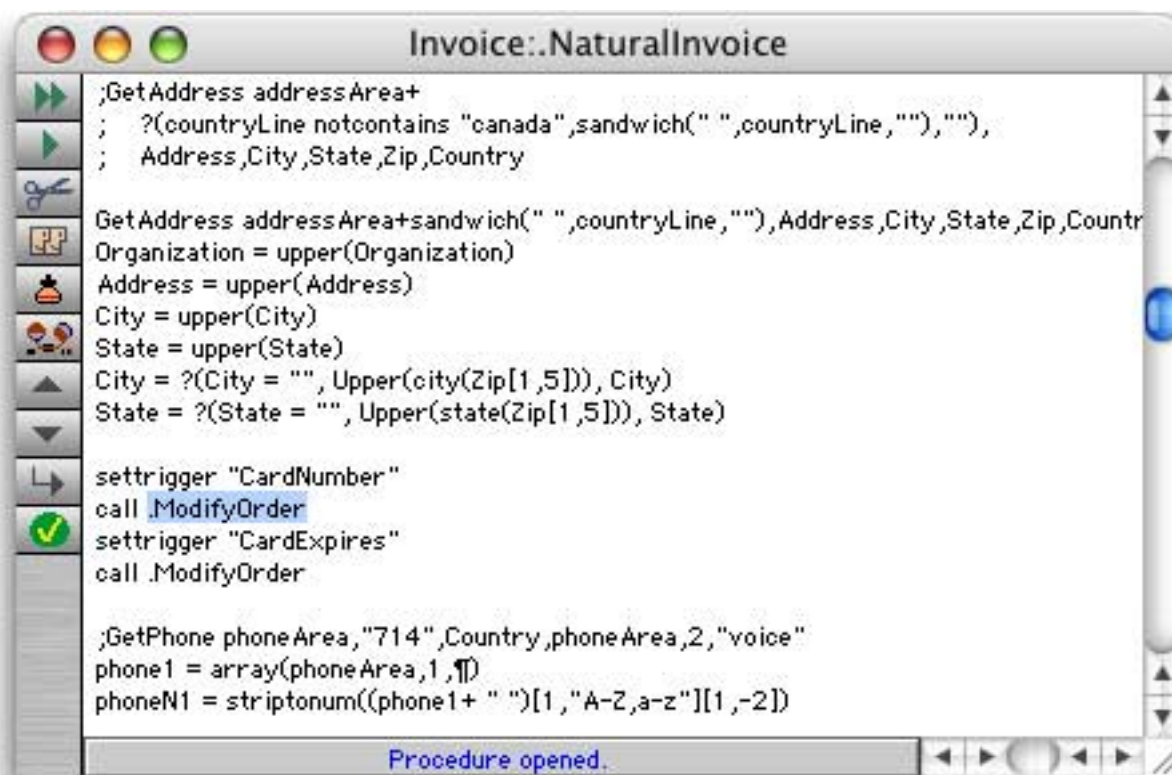
double click to open item

If the database is already open, the form, procedure or crosstab will open in a new window. If the database is not already open, the wizard will ask you if you want to open it.



If you press **Open Normal**, the database will open normally. Any saved window positions will open, and if the database has an .Initialize procedure it will be triggered (just as if you had double clicked on this database on your desktop, or opened it from the Open File dialog.) If you press **Open Secret**, the wizard will not open any saved windows, and it will not trigger the .Initialize procedure, if any. Only the requested form, procedure or crosstab will open.

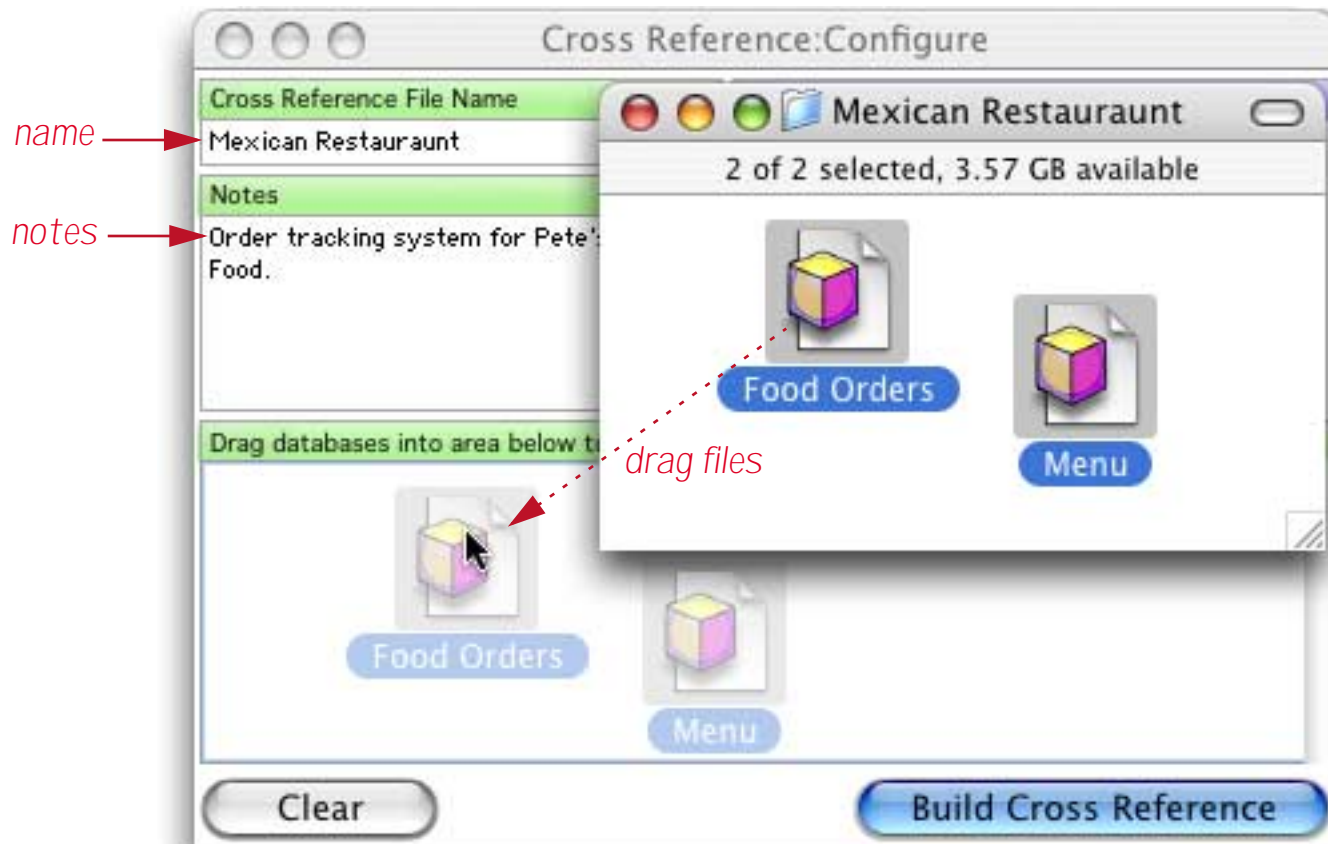
If you double clicked on a procedure, the wizard will automatically highlight the first occurrence of the text you originally searched for.



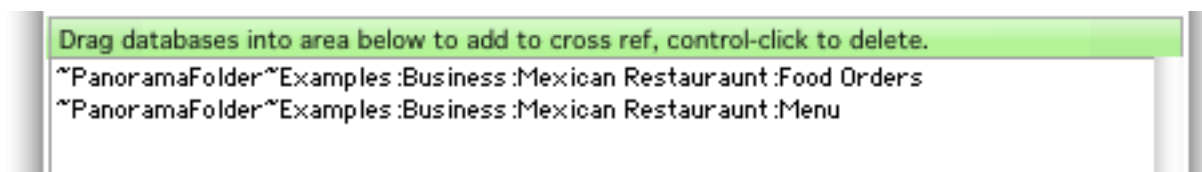
You can use the Find Next command to locate additional occurrences of this text.

Setting up a New Cross Reference

To set up a new cross reference, click on the **Configure** button. You'll see the current configuration in the Configuration window. Since you want to create a new configuration, press the **Clear** button, then type in the name for the new cross reference you are going to create (up to 27 characters). You can also optionally type in some notes. Finally, drag the files you want to be included in the cross reference into the bottom portion of the window.



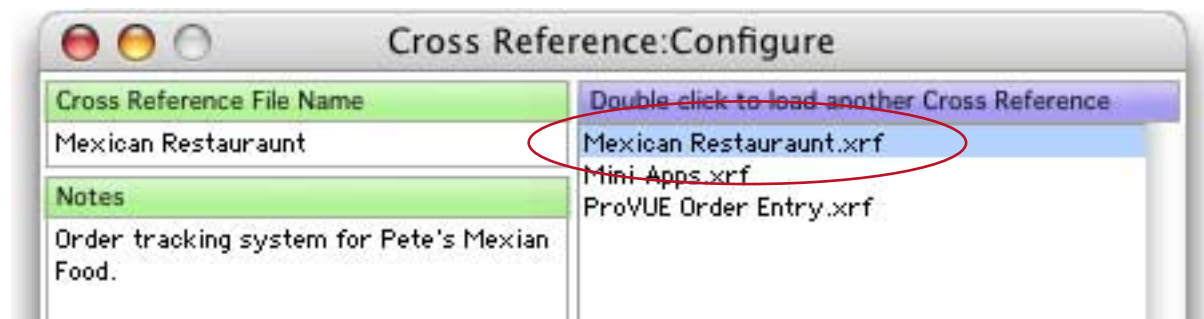
When you release the mouse the databases will be listed in the configuration window. If necessary you can drag additional files into this area.



When everything is set up, press the **Build Cross Reference** button. After a delay, the Configuration window will close and the new cross reference is ready to use. (The delay may be from a few seconds to a minute, depending on the complexity level of the databases involved). You can now search the cross reference as described in the previous section.



If you press the **Configure** button you'll see your new cross reference listed.



Updating a Cross Reference

A cross reference is a snapshot in time that reflects the contents of your database at the time you pressed the **Build Cross Reference** button. As you work on your databases the cross reference will gradually go out of date. When that happens you can update the cross reference by pressing the **Configure** button, then the **Build Cross Reference** button.

50 Ways to Trigger a Procedure

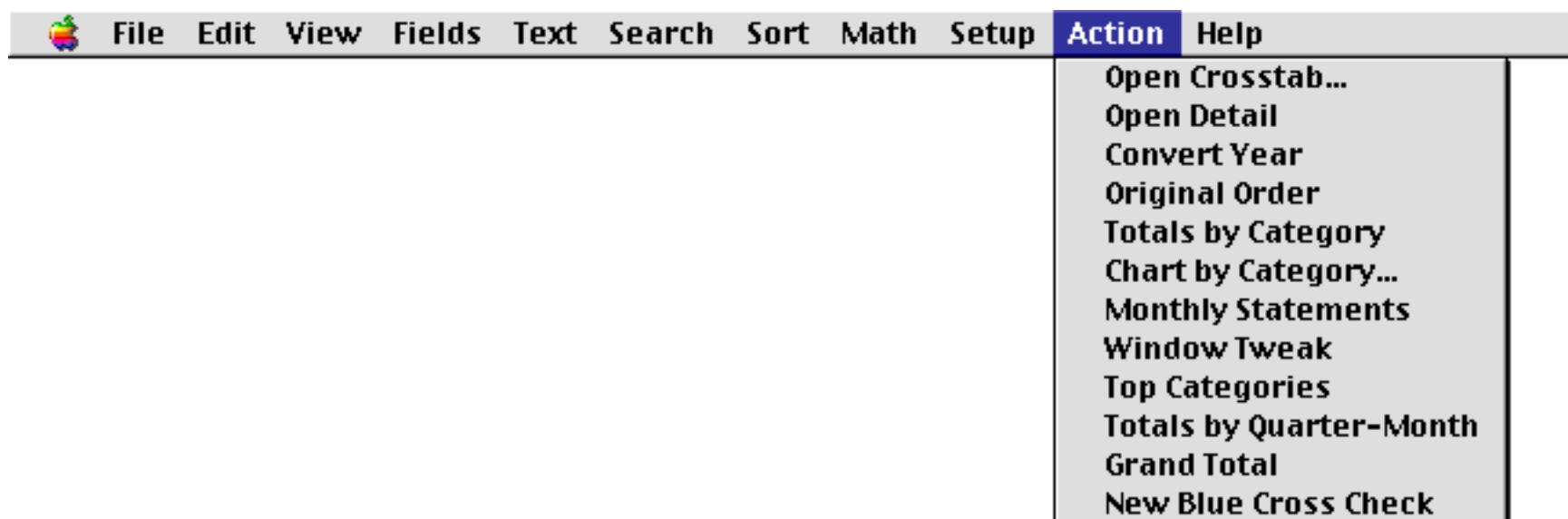
Procedures don't start up on their own — they must be triggered somehow. There aren't really 50 ways to trigger a procedure, but there are quite a few.

There are basically two types of triggers that can activate a procedure: **explicit triggers** and **hidden triggers** (implicit). Explicit triggers allow the user to deliberately trigger a procedure, for example by pressing a button or choosing an item from a menu. Hidden triggers activate a procedure automatically when the user performs some normal Panorama action. Examples of user actions that can cause a hidden trigger to activate include adding new records to a database, deleting records, opening a file, closing a window and many more. Procedures that are activated by hidden triggers can customize the way Panorama responds to these user actions, giving the programmer tremendous flexibility in creating a user interface that is appropriate for the task at hand.

The same procedure can be triggered different ways at different times. For example, the same procedure could be triggered both by a menu command or a button. If necessary, a procedure can use the `info("trigger")` function to find out how it was triggered.

The Action Menu

The **Action** menu is the simplest way to allow a procedure to be triggered. All you have to do is create the procedure, and it is automatically listed in the **Action** menu. The **Action** menu is added to the end of the standard menus, and the user can activate any procedure simply by selecting its name from the **Action** menu. (Advanced tip: If a procedure may be triggered other means in addition to the **Action** menu (for example, by a button), you can use the `info("trigger")` function to find out which way the procedure was triggered. If the procedure was triggered by the menu this function will return **Action Menu.**)



Panorama allows some variations from the basic one-size-fits-all **Action** menu. The programmer can give the **Action** menu a different name, or even split the **Action** menu into multiple menus. The programmer can also exempt some procedures so that they are not listed in the **Action** menu (a procedure that is not listed can only be triggered some other way, for example by a button).

The **Action** menu does have some significant limitations. **Action** menus can only be added to the standard menus, they cannot replace the standard menus. **Action** menus cannot have any submenus. The **Action** menu cannot change when the user switches from form to form—it always contains the same items (unless you switch to a different database). In addition, there must be a separate procedure for each menu item in the **Action** menu. It is not possible to have multiple menu items handled by a single procedure. With these restrictions in mind, the **Action** menu is by far the easiest way to set up your own menus in a Panorama database. For many custom Panorama databases, the **Action** menu is the only user interface. (Note: Prior to version 3.0 the **Action** menu was called the **Macro** menu.)

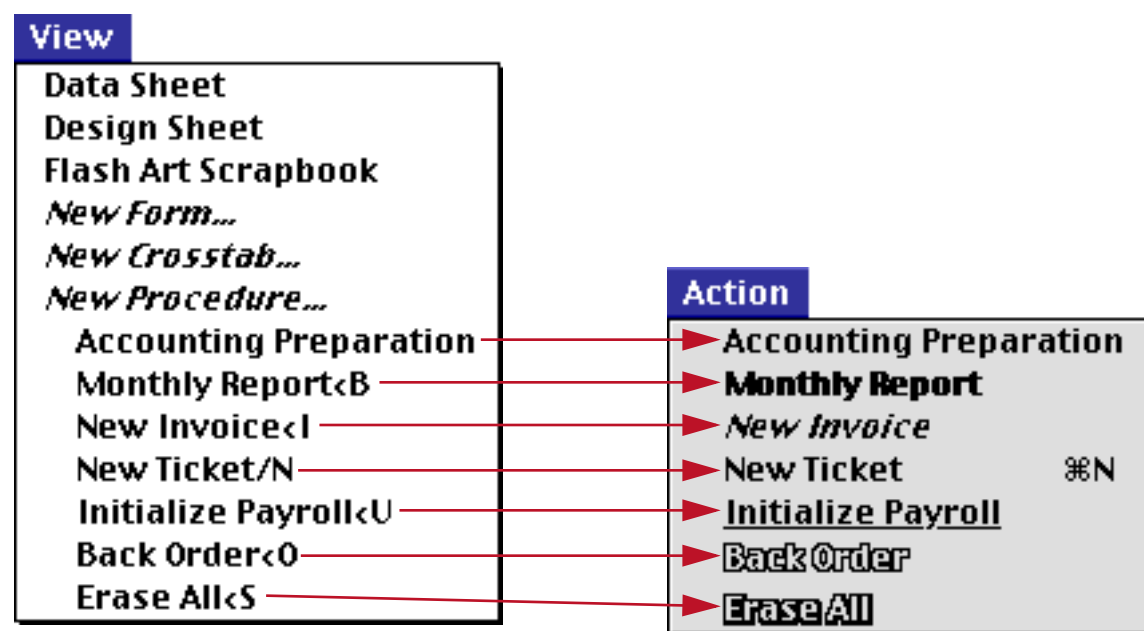
Action Menu Options

By adding special characters to a procedure's name, you can change the way the procedure is displayed in the **Action** menu, or even remove the procedure from the menu completely.

There are two special characters that should never be used in a procedure name that is listed in the **Action** menu: **^** and **;**.

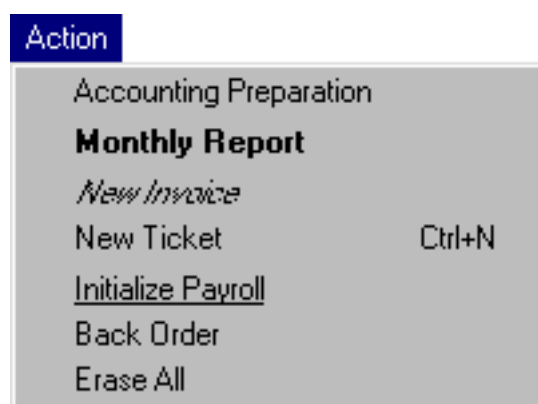
Setting Different Menu Item Styles (Bold, Italic, etc.)

You can make a procedure name appear in several different styles in the **Action** menu—bold, italic, underline, outline, shadow, or a combination of these styles. To change the style of a menu item you must add a special suffix to the end of the procedure name. The suffix consists of the **<** character followed by the letter **B** (bold), **I** (italic), **U** (underline), **O** (outline) or **S** (shadow). The action menu below show all six different styles (including plain) and the procedure names for creating those styles.



You can also combine styles with multiple suffixes, for example **Initialize Payroll<B<I** for both italic and bold. You can also combine a style with a command key equivalent, for example **Back Order<I/B**.

Here's the same menu on a Windows based system.



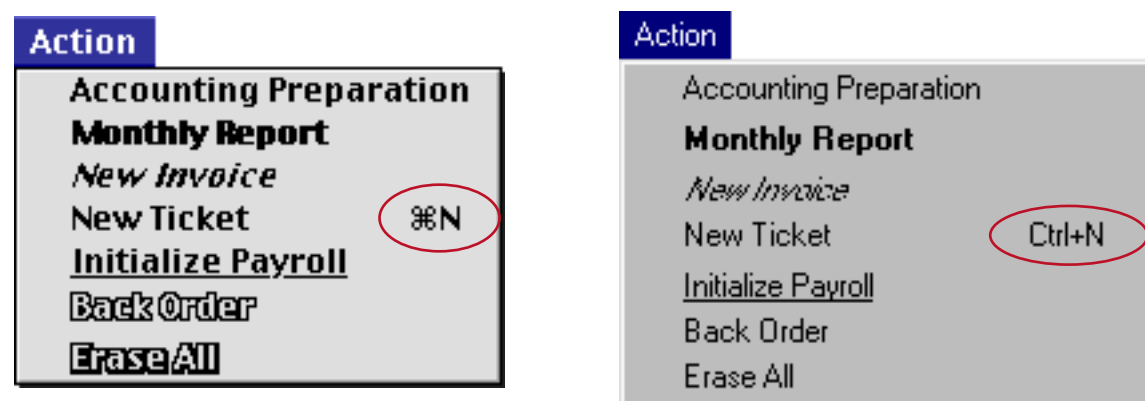
As you can see, the Outline and Shadow styles do not appear on Windows systems.

Shortcuts/Command Key Equivalents

Like other menu items, procedures in the **Action** menu can have keys on the keyboard assigned to them. On the Macintosh these are called **Command Key Equivalents**, on the PC (Windows) they are called **shortcuts**. To assign a key to a procedure you add a suffix consisting of a **/** character followed by the key you want to assign to the procedure. For example, a procedure named

New Ticket/N

will show up in the **Action** menu assigned to the **N** key. Here is what this menu looks like on both the Mac (left) and Windows (right).



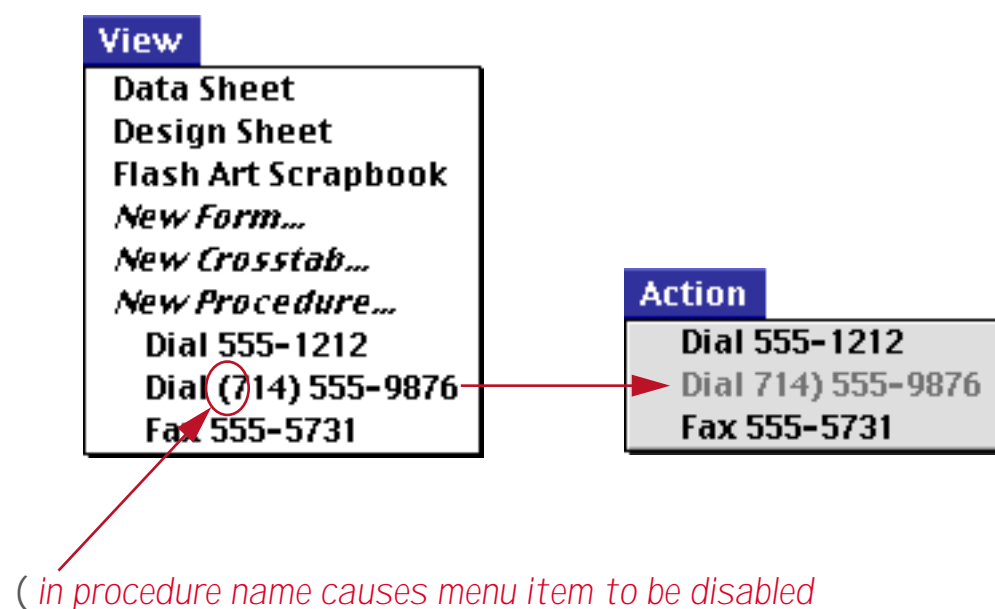
You can run this procedure by choosing it from the menu, or by pressing **Command-N** on the Macintosh or **Control-N** on a Windows based computer.

If a procedure's key assignment conflicts with one of Panorama's standard key assignments, the procedure will override the standard equivalent. For example, **Command/Control-P** is normally a command key equivalent for **Print**, but if you add a procedure called **Post Checks/P**, pressing **Command/Control-P** will trigger the procedure instead of printing.

Note: You cannot assign a command key equivalent to an "unlisted" procedure (one that begins with a period). Only procedures that appear in the **Action** menu can have command key equivalents.

Disabled Menu Items

If a procedure name contains the (character, the procedure name will appear in the menu but will be disabled (gray).

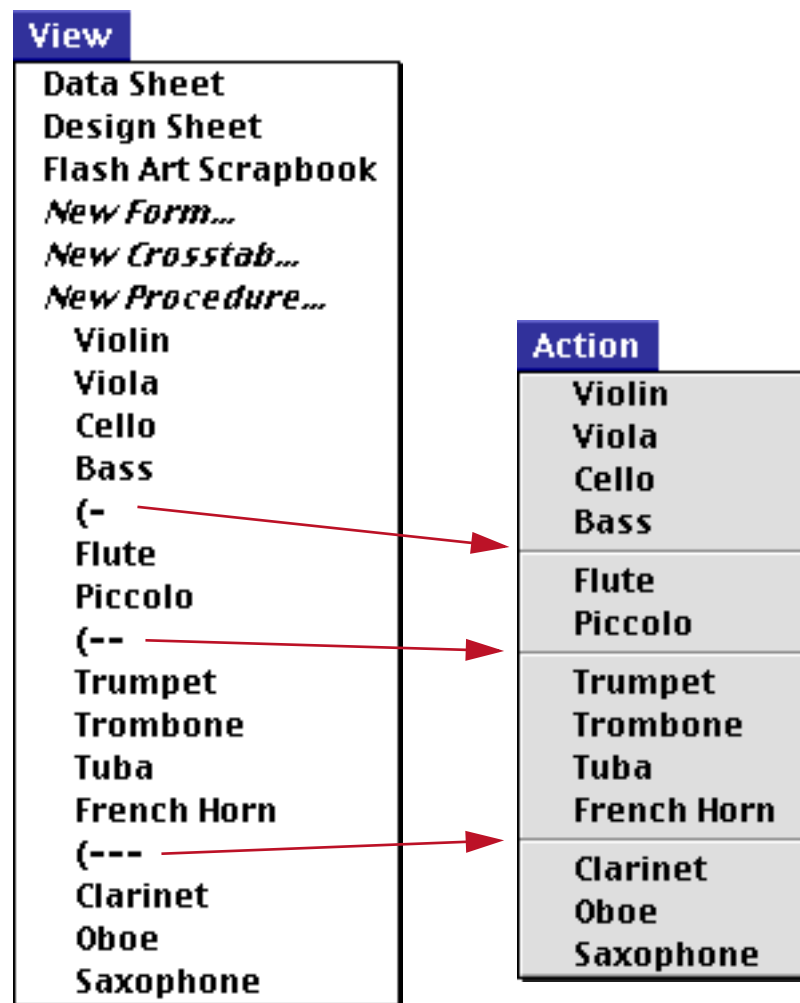


Don't use parenthesis in a procedure name unless you want the procedure to be disabled.

Separator Lines in a Menu

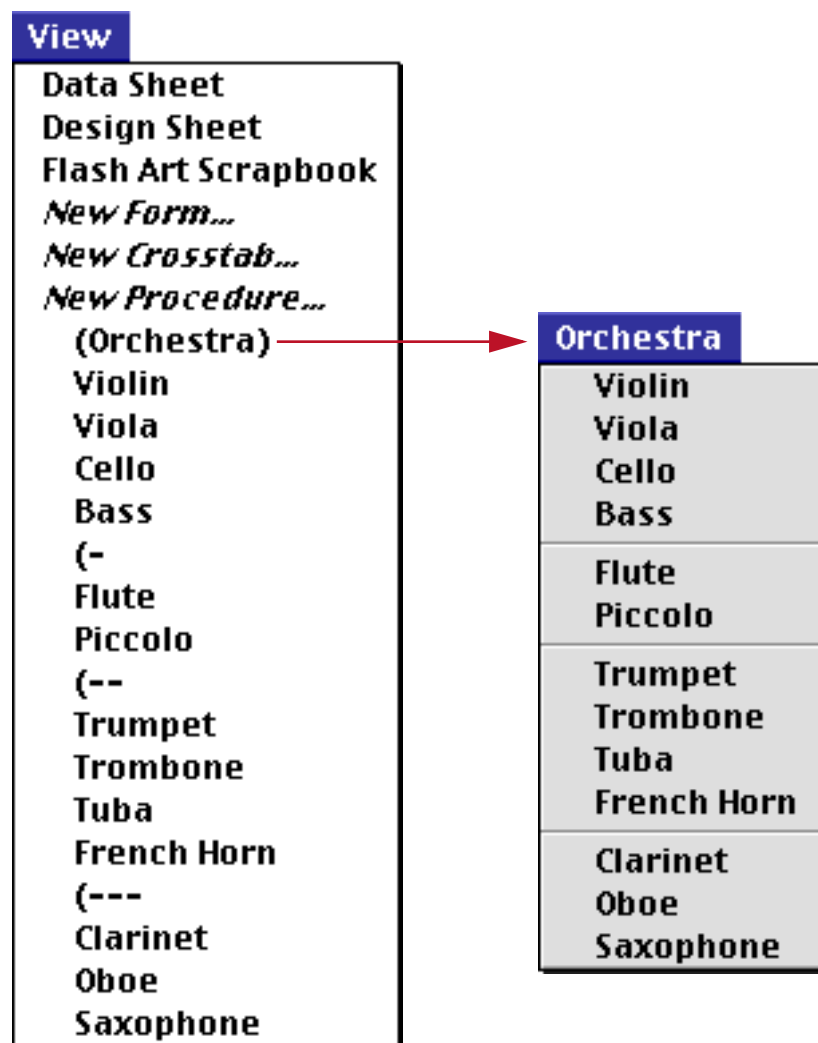
Many menus contain one or more gray lines separating different sections in the menu. To add a gray line to the **Action** menu, create a procedure with a name that start with (-. Use the **New Procedure** command in the **View** menu to create the procedure. Since the procedure will be disabled in the menu, it should not contain any statements.

To add a second gray line to the menu, create a procedure named (--. The third gray line should be named (---, the fourth (----, etc. (Each procedure name must contain a different number of dashes because Panorama does not allow duplicate procedure names.) Here's an example of an **Action** menu divided into four sections.



Renaming the Action Menu

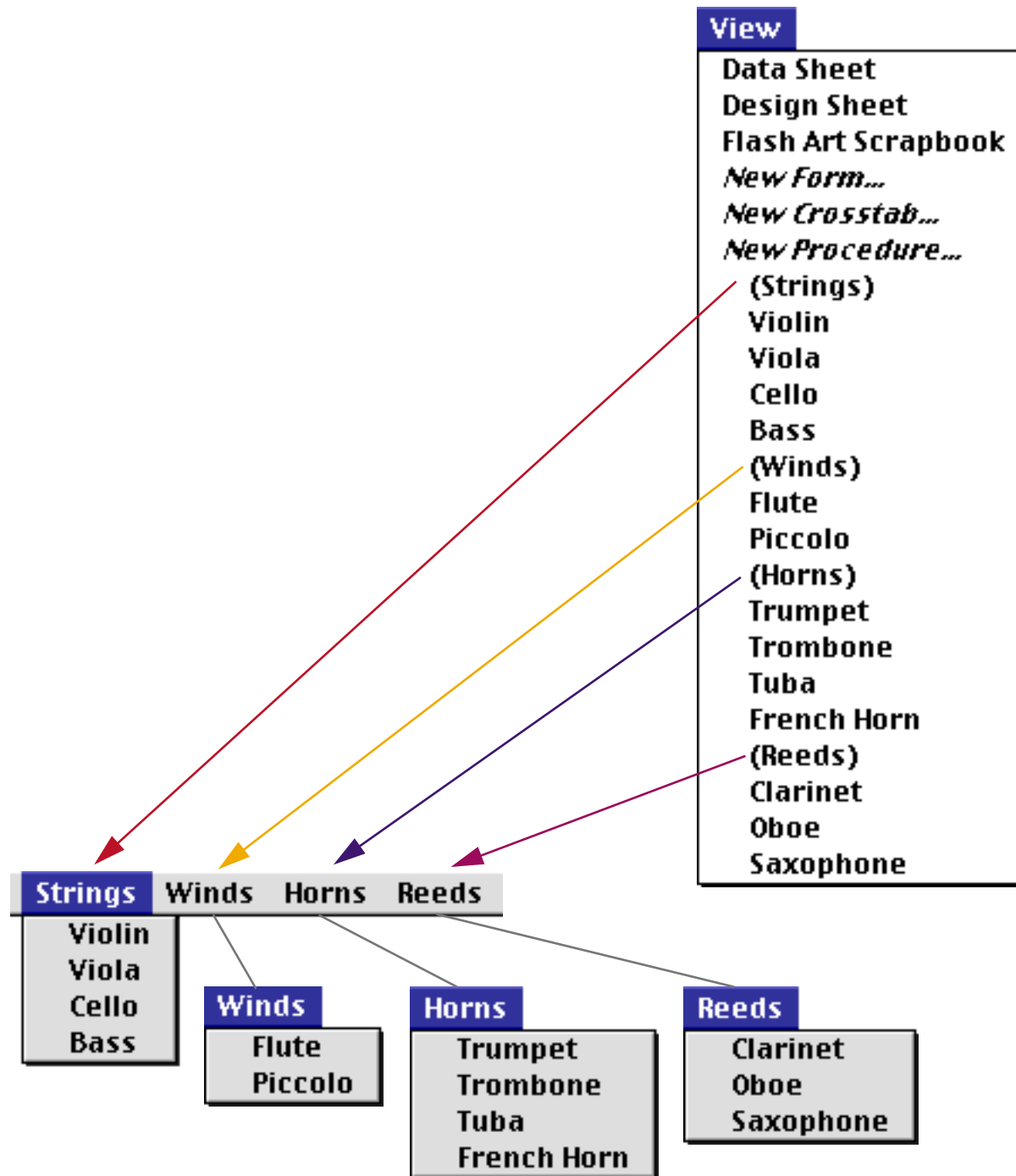
To give the **Action** menu a different name, insert an empty procedure with a name that begins and ends with a parenthesis as the very first procedure in the database. For example, inserting a procedure named **(Orchestra)** before the first procedure causes the **Action** menu to become the **Orchestra** menu.



See "[Creating a New Form, Crosstab or Procedure](#)" on page 182 to learn how to insert a procedure in any position.

Dividing the Action Menu into Multiple Menus

If your database contains lots of procedures you may want to split the **Action** menu into two or more separate menus. To split the **Action** menu into separate pieces, insert an empty procedure with a name that begins and ends with parentheses. For example, to start a new menu named **People** add a new procedure named **(People)**. All of the procedures below this point will be listed in the People menu. You may split the **Action** menu into up to 12 separate menus. The example below shows an action menu split into four different menus.

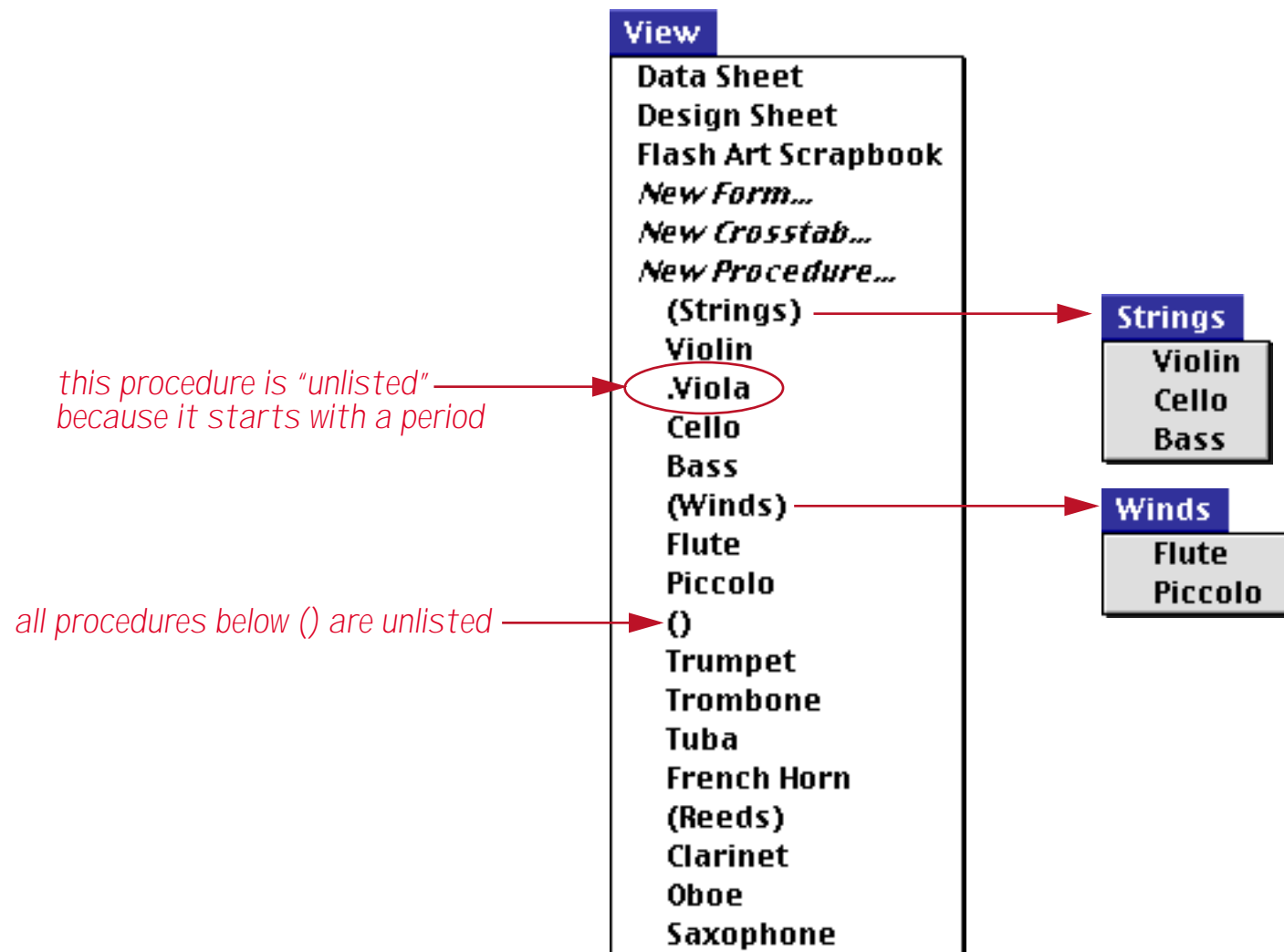


If necessary you can re-arrange the procedures to organize them into menus. See [“Changing the Order of Forms, Crosstabs or Procedures”](#) on page 183.

“Unlisted” Procedures

You may not want the **Action** menu to list the special procedures you create for buttons, automatic events, custom menus, or subroutines. To keep a single procedure out of the menu, add a period to the beginning of the procedure name. Any procedure name that begins with a period will be “unlisted,” for example [.Balance](#) or [.Prepare Chart](#).

To keep an entire group of procedures out of the menu insert a menu named (). Any procedure below a procedure named () will not appear in any menu.



Live Menus

The **Action Menu** is a very simple method for adding menus to your database. If you need more flexibility, however, you can use Panorama's "Live" menu feature. Live menus give you complete control over the content and appearance of each menu.

The secret behind the Live Menu system is a new special variable named **LiveMenuFormula**. If Panorama finds this variable it uses the contents to control the arrangement and content of the menus in the menu bar. This variable can be either a fileglobal variable (in which case it controls the menus for all windows in the current database) or a windowglobal variable (in which case it will control only the current window). In most cases, however, you will not manipulate this variable directly, but will access it with either the **FileMenuBar** or **WindowMenuBar** statement.

The FileMenuBar Statement

The **FileMenuBar** statement sets up the menu configuration for every window in the current database. Typically you might use this statement in the **.Initialize** procedure so that the menus will be set up as the database opens, but you can use it at any time to change the configuration. The statement has two parameters.

```
FileMenuBar StandardMenus,CustomMenus
```

The **StandardMenus** parameter is simply a list of the standard menus you want to include at the beginning of the menu bar. Choices available are **APPLE**, **FILE**, **EDIT**, **VIEW**, **TEXT**, **SEARCH**, **SORT**, **MATH** and **SETUP**. (Note: none of these choices are case sensitive.) You can also disable the action and wizards menu by using **-ACTION** and/or **-WIZARDS**. Items may be separated by spaces or commas. To include all standard menus, use **ALL**. To include the basic standard menus (Apple, File and Edit) use **BASIC**. (Note: On PC systems, the **Apple** menu is always excluded even if you ask it to be included.) If all you want to do is suppress some of the standard menus you can use this parameter and leave the **CustomMenus** parameter blank. For example, if you want just the **Apple**, **File**, **Edit**, **Action** and **Wizard** menus, use this statement.

```
FileMenuBar "Basic", ""
```

If you want the **Apple**, **File**, **Edit**, **View** and **Action** menu, use this statement.

```
FileMenuBar "Apple File Edit View -Wizards", ""
```

The **CustomMenus** parameter contains the formula for your live menus. There are a number of techniques for writing this formula, but the easiest is to use the **menu()** and **menuitems()** functions, like this:

```
FileMenuBar "Basic",  
    menu("Colors")+menuitems("White;Yellow;Orange;Red;Green;Blue;Violet;Black")
```

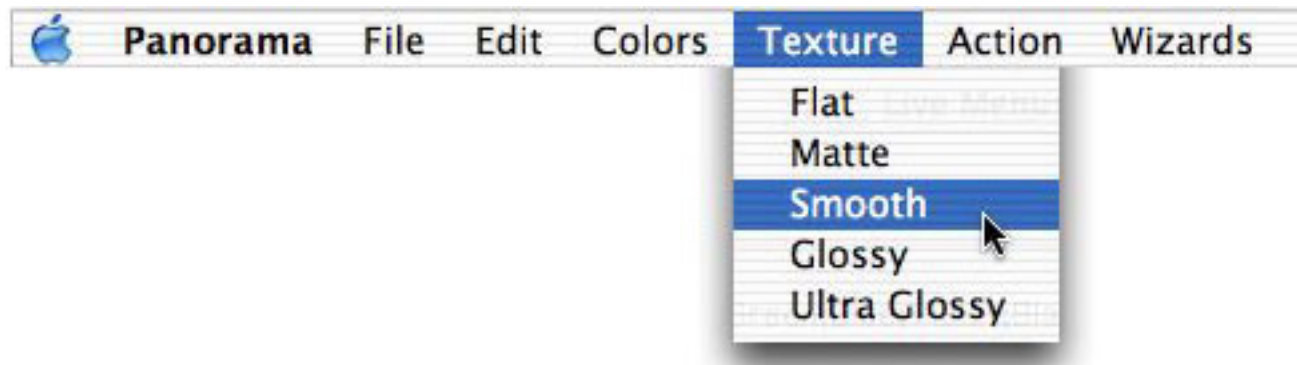
As you have probably guessed, this creates a custom menu named **Colors** with eight items.



You can repeat to add as many menus as you like:

```
FileMenuBar "Basic",
  menu("Colors")+menuitems("White;Yellow;Orange;Red;Green;Blue;Violet;Black;")+
  menu("Texture")+menuitems("Flat;Matte;Smooth;Glossy;Ultra Glossy;")
```

This creates two menus — **Colors** and **Texture**:

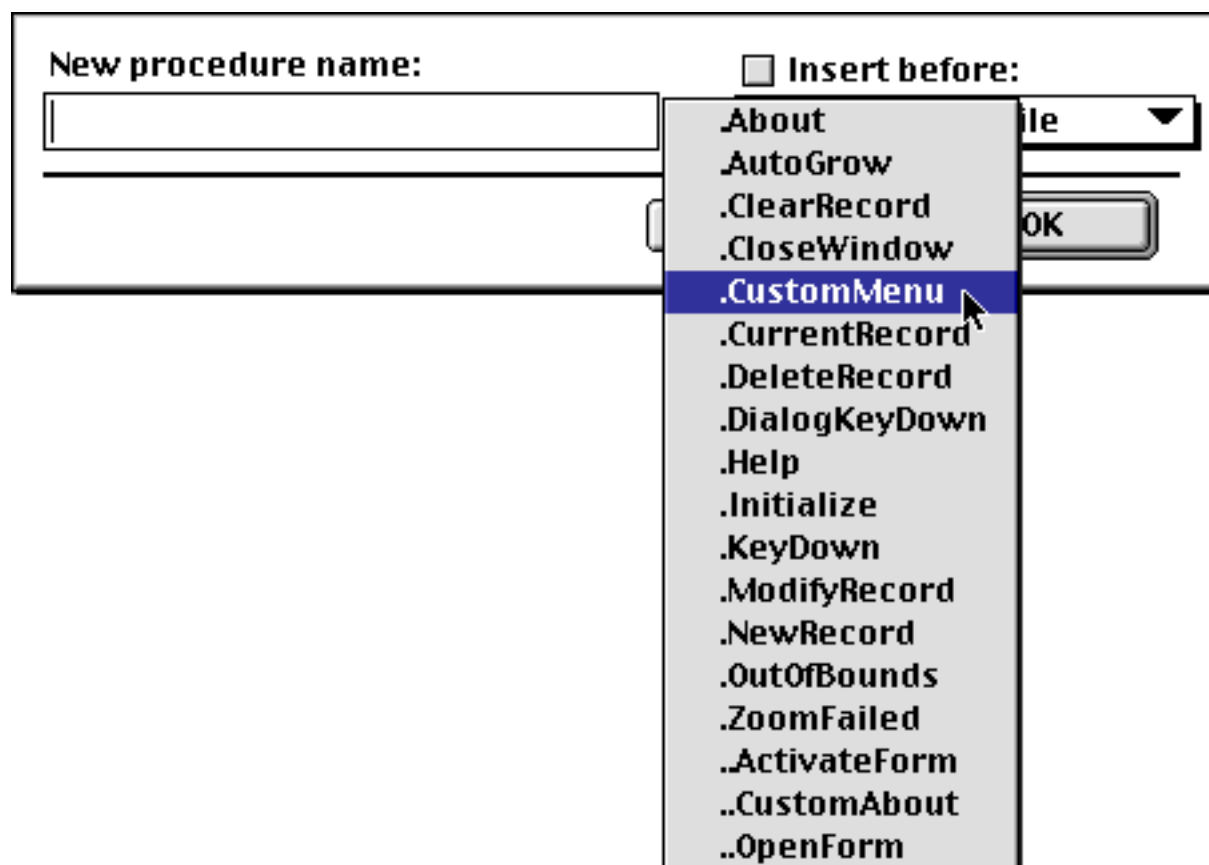


Remember that you must put these statements into a procedure and run the procedure before the live menus will appear. Until that point Panorama's standard menus will be used. To make sure that the live menus are used immediately when the database is opened you should put the `FileMenuBar` statement into the `.Initialize` procedure (see "[.Initialize](#)" on page 382).

The `.CustomMenu` Procedure

What happens when a user pulls down a custom menu and selects a menu item? Choosing an item in a custom menu automatically starts a special procedure. This procedure must be called `.CustomMenu`. If the database does not have a `.CustomMenu` procedure then you'll still be able to pull down custom menus, but they won't do anything.

(Tip: When you create the `.CustomMenu` procedure, make sure that the procedure name is spelled and capitalized correctly. Don't forget the period at the beginning. The easy way to do it right is to select `.CustomMenu` from the pop-up menu in the `New Procedure` dialog (see "[Creating Hidden Trigger Procedures](#)" on page 378).



If the **.CustomMenu** procedure name is not spelled correctly, Panorama won't be able to find and trigger the procedure when a custom menu item is used, and your custom menus won't work.)

Programming the .CustomMenu Procedure

Since the **.CustomMenu** procedure is triggered for all custom menu items, the procedure needs a way to figure out what item was chosen and act accordingly. For example, if the user pulls down a menu item you've created called **Sort by City**, you'll want something different to happen then if the user selects **Void Transaction**.

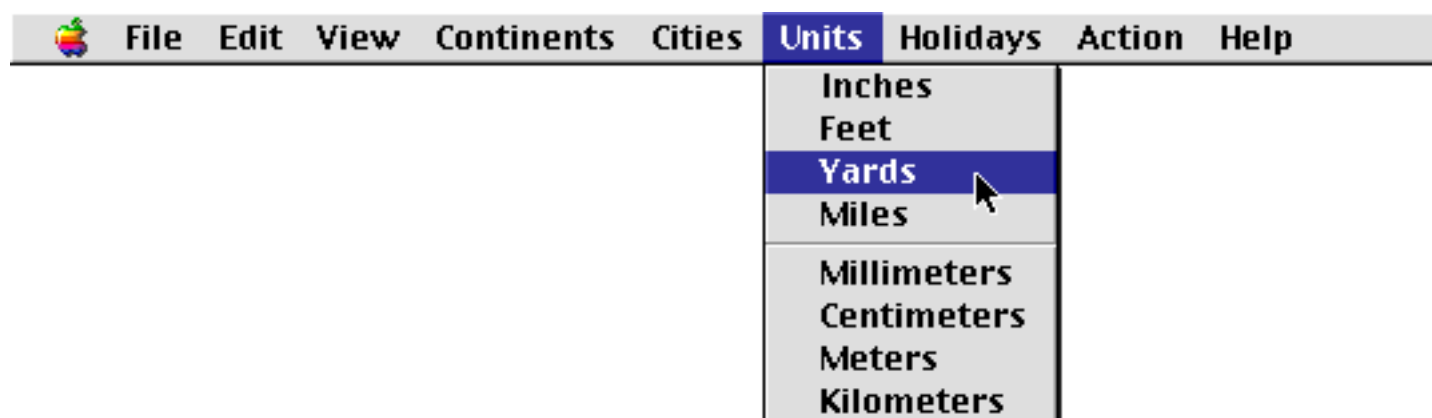
Whenever a custom menu item is chosen, Panorama stores the name of the custom menu and the name of the item. The programmer can retrieve this information using the `info("trigger")` function (see "[INFO\("TRIGGER"\)](#)" on page 5433 of the *Panorama Reference*). By combining the `info("trigger")` function with **if** or **case** statements (see "[IF Statements](#)" on page 257 and "[CASE Statements](#)" on page 259) the programmer can create a **.CustomMenu** procedure that performs the correct action for every custom menu item. The following sections will illustrate several methods for programming **.CustomMenu** procedures to operate correctly.

The info("trigger") Function

The `info("trigger")` function can be used by any procedure to find out how that procedure was triggered. If the procedure was triggered by a custom menu, the `info("trigger")` function will return the word **Menu** followed by the menu name and menu item name, separated by periods.

```
Menu.<Menu Name>.<Menu Item Name>
```

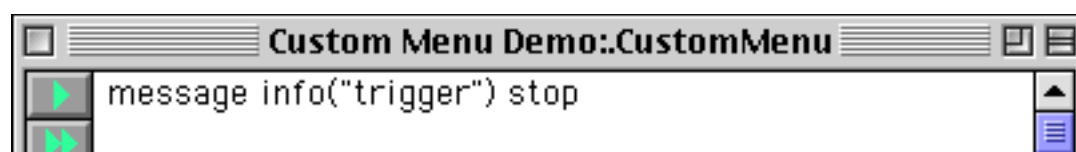
For example, suppose you select **Yards** from the **Units** menu.



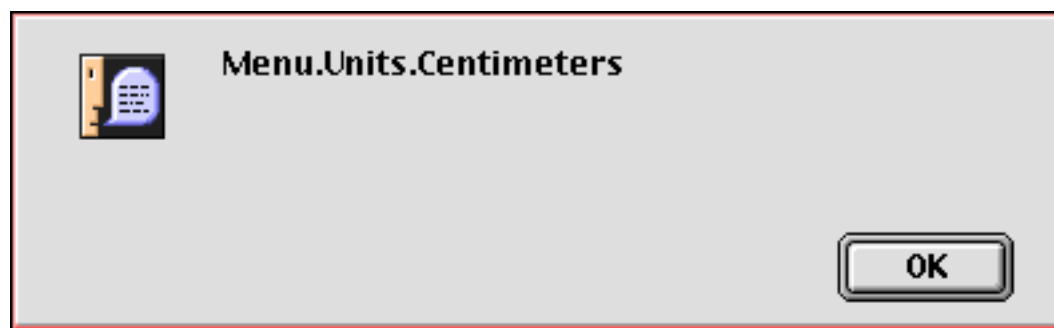
When this item is chosen the `info("trigger")` function will return the value:

```
Menu.Units.Yards
```

If you are ever in doubt about what value the `info("trigger")` function will contain for a menu item, temporarily insert the following line into the top of the **.CustomMenu** procedure.



Now choose the custom menu item in question. An alert will appear showing you the exact value produced by the `info("trigger")` function.



Once you have the value, be sure to go back and remove the temporary line from `.CustomMenu` procedure. A handy way to do this is to comment it out so that it can be easily re-activated later (see “[Commenting Out Statements](#)” on page 304).

Processing Custom Menus with Simple IF's

The simplest way to process custom menus is to use the `if` statement (see “[IF Statements](#)” on page 257). In this technique a similar block of statements is repeated over and over, once for each custom menu item. Each block starts with an if statement that uses `info("trigger")` to decode the menu item name. Then there are one or more statements that perform the actual operations for this menu item. Since we don't want any further actions for other menu items to be performed, this is followed by a `stop` statement (see “[Stopping the Program](#)” on page 278). The `endif` statement terminates the entire block.

```
if info("trigger") = "Menu.<Menu Name>.<Item Name>"
    statement1
    statement2
    statement3
    ...
    stop
endif
```

The `.CustomMenu` procedure should contain one of these blocks for each custom menu item. Since each block of statements is completely self contained, the blocks can be in any order you want. The example below shows a `.CustomMenu` procedure written for two custom menus with two menu items apiece.

```
if info("trigger") = "Menu.Organize.SortByName"
    field LastName
    sortup
    field FirstName
    sortupwithin
    stop
endif
if info("trigger") = "Menu.Organize.SortByZip"
    field Zip
    sortup
    stop
endif
if info("trigger") = "Menu.Transaction.Add"
    AddRecord«
    stop
endif
if info("trigger") = "Menu.Transaction.Void"
    Description="Void"
    Amount=0
    stop
endif
```

Processing Custom Menus with Nested IF's

If your database has a lot of custom menu items, the technique described in the last section can be slow for items that are processed toward the bottom of the **.CustomMenu** procedure. There may be a noticeable delay as Panorama processes all the **if** statements. The solution to this delay is to group the blocks together by menus using nested **if** statements. For example, suppose your database uses 6 custom menus with 15 items apiece. Using the simple **if** statement technique there could be a delay of as many as 90 **if** statements before the statements that actually do the work get started. Using nested **if** statements this delay is reduced to a maximum of 21 **if** statements.

The example below shows the previous example rewritten to use nested **if** statements. The outer level of **if** statements selects what menu is being processed, while the inner level selects the individual menu items.

```
if info("trigger") beginswith "Menu.Organize."
  if info("trigger") endswith ".SortByName"
    field LastName sortup
    field FirstName sortupwithin
    stop
  endif
  if info("trigger") endswith ".SortByZip"
    field Zip sortup
    stop
  endif
endif
if info("trigger") beginswith "Menu.Transaction."
  if info("trigger") endswith ".Add"
    AddRecord
    stop
  endif
  if info("trigger") endswith ".Void"
    Description="Void"
    Amount=0
    stop
  endif
endif
endif
```

Splitting the Trigger into Menu/Item Names

In some cases it may be advantageous to split the value returned from **info("trigger")** back into separate menu and menu item names. This can be done with the **array()** function as shown in the example below (see [“Text Arrays”](#) on page 93).

This example assumes that the database contains a field called **Carrier**, and a custom menu called **Airlines** that contains menu items listing airlines: **American**, **Delta**, **Southwest**, etc. When the user selects an airline the name of the airline is copied into the **Carrier** field.

```
local MenuName,MenuItemName
MenuName=array(info("trigger"),2, ".")
MenuItemName=array(info("trigger"),3, ".")

if MenuName="Airlines"
  Carrier=MenuItemName
  stop
endif
/* other menu processing continues below */
...
```

An even easier method for doing this is to use the **splitmenutrigger** statement. See the **Programming Reference** wizard for more information on this statement.

Menus with Modifier Keys

Sometimes you may want to have a custom menu item perform a different action if a modifier key is pressed (**Shift**, **Control**, **Option**, **Command** or **Alt**). The program can test for these modifiers with the `info("modifiers")` function. This function returns the names of all the modifier keys that are pressed down.

The partial example below uses the `info("modifiers")` function to create a shortcut for the **Void** menu item. This procedure is programmed so that a confirmation alert normally appears before the transaction is voided, but if the user holds down the **Option** key the alert is skipped (**Alt** key on PC systems).

```
if info("trigger") = "Menu.Transaction.Void"
  if (not info("modifiers") contains "option")
    alert 1014,"Are you sure you want to void this transaction"
    if info("dialogtrigger")="No"
      stop
    endif
  endif
  Description="Void"
  Amount=0
  stop
endif
```

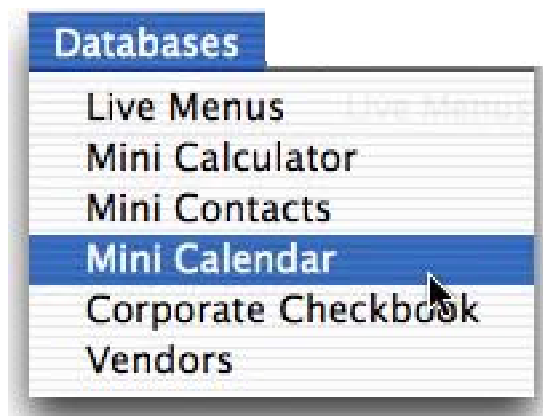
Since the user has no way to tell that a modifier key affects the operation of a menu item, this technique should be used with care. Don't make pressing a modifier key cause a completely different operation. In general this technique should only be used for slight variations (like the shortcut above), or to allow for secret undocumented operations that you don't want someone to stumble across accidentally.

Building Menus from Arrays

You'll often want to build a menu (or part of a menu) from an array. The `arraymenu()` function makes this easy to do. This function turns a carriage return delimited array into a series of menu items. This example builds a menu that lists all of the currently open databases.

```
FileMenuBar "Basic",menu("Databases")+arraymenu(info("files"))
```

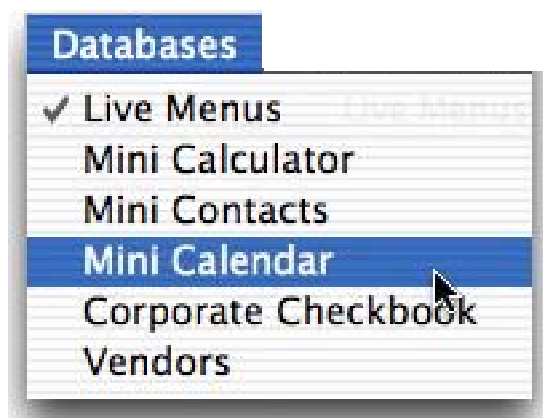
This menu will automatically adjust as databases are opened and closed.



The `checkedarraymenu()` function is similar, but allows one of the menu items to be checked. The second parameter specifies what item should be checked.

```
FileMenuBar "Basic",menu("Databases")+checkedarraymenu(info("files"),info("databasename"))
```

In this case, the currently opened database is checked.



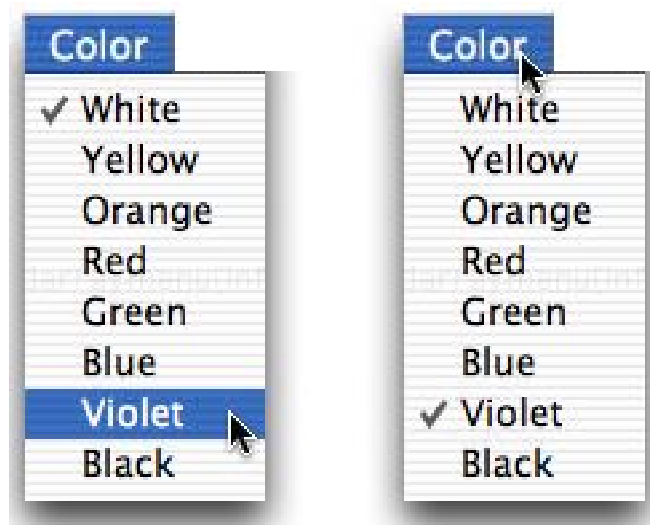
A more useful example uses a menu to set a variable, in this case allowing a color to be selected from a list of colors. To create the menu place the following statements in the `.Initialize` procedure:

```
FileGlobal colorOptions,colorChoice
colorOptions=replace("White;Yellow;Orange;Red;Green;Blue;Violet;Black",";","")
define colorChoice,"White"
FileMenuBar "Basic",menu("Color")+checkedarraymenu(colorOptions,colorChoice)
```

Here's the code that must be placed in the `.CustomMenu` procedure to actually operate the menu (see "[The .CustomMenu Procedure](#)" on page 363).

```
if info("trigger") beginswith "Menu.Color."
    colorChoice=array(info("trigger"),3,".")
endif
```

That's all there is too it. Now when an item is selected from the **Color** menu, the item will be checked.



The value of the choice is in the `colorChoice` variable, where it can be used by other formulas or procedures.

Command Key Equivalents

Within the `menuitems` function an item can be followed by the `-` character and one or more options. Everything before the `-` is the menu name, with the options after. The `/` option is used to set up command key equivalents. This example creates three menu items with corresponding equivalents.

```
FileMenuBar "Basic",
    menu("Shipping")+menuitems("Mail-/M;UPS-/U;FedEx-/F")
```


To select **Mail** with the keyboard, type **Command-M** (**Control-M** on the PC). **UPS** is **Command-U/Control-U** while **FedEx** is **Command-F/Control-F**.



Menu Styles

Using the < option, menus can be bold, italic, underline, or a combination of these three choices.

```
FileMenuBar "Basic",
    menu("Currency")+menuitems("Dollar<<B<U/D;Euro;Franc<<I;Lira<<I;Peso;Pound<<U;Yen")
```

Here is the menu created by this statement.



Disabled Menu Items and Separator Lines

To disable a menu item make the first character in the menu name a left parenthesis (.

```
FileMenuBar "Basic",
    menu("Currency")+menuitems("Dollar;Euro;(Franc;(Lira;Peso;Pound;Yen")
```

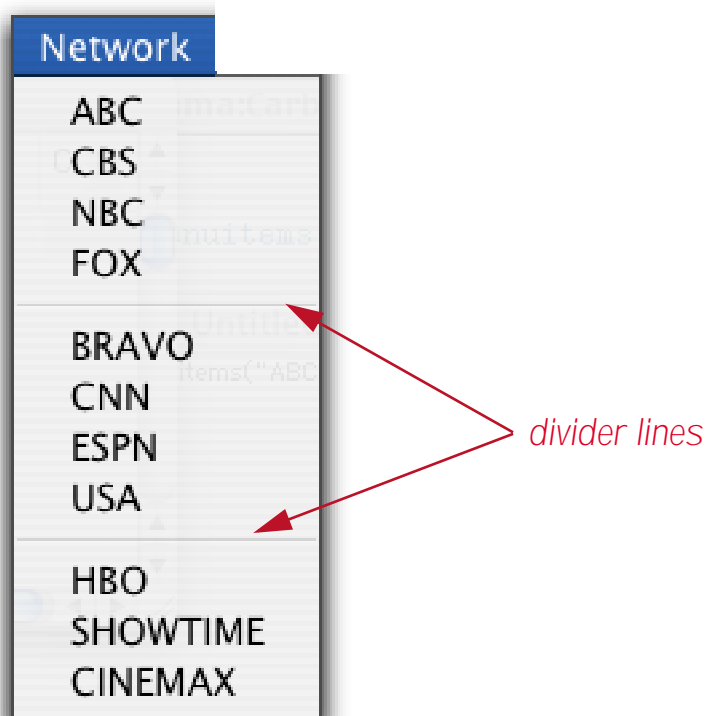
Here is the menu with the disabled items.



To create a divider line the menu item should be (-, like this:

```
FileMenuBar "Basic",
    menu("Network")+
    menuitems("ABC;CBS;NBC;FOX;(-;BRAVO;CNN;ESPN;USA;(-;HBO;SHOWTIME;CINEMAX")
```

This menu is divided into three sections: Broadcast networks, basic cable, and premium cable:

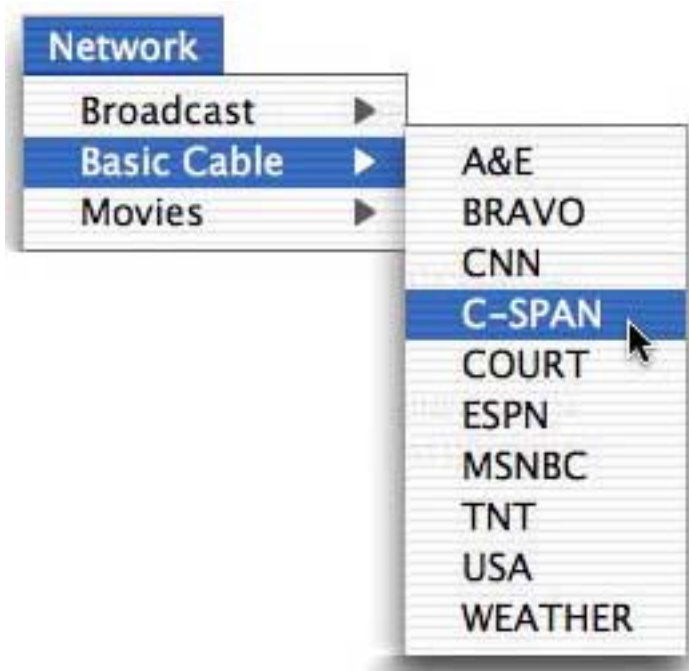


Submenus (Hierarchical Menus)

One of the advantages of the Live Menu system is that it makes it easy to attach a submenu to any menu item (even an item in a submenu for multiple menu levels of submenus. If you use submenus they must be defined before they are used. Submenus are defined with the `submenu()` function instead of the `menu()` function. To use a submenu you must place (`submenu name`) in the options area at the end of the menu item (after the `->` character).

```
FileMenuBar "Basic",
  submenu("Broadcast")+menuitems("ABC;CBS;NBC;FOX;WB;")+
  submenu("Basic")+menuitems("A&E;BRAVO;CNN;C-SPAN;COURT;ESPN;MSNBC;TNT;USA;WEATHER;")+
  submenu("Premium")+menuitems("HBO;SHOWTIME;CINEMAX;")+
  menu("Network")+menuitems("Broadcast->(Broadcast);Basic Cable->(Basic);Movies->(Premium)")
```

This example creates a menu with three submenus.



Multiple Column Menus

On MacOS, a menu that has too many items to fit within the height of the display will scroll. By using the `columnmenu()` function you can create a menu that wraps to two , three, four or more columns as needed. Otherwise this function is the same as the `menu()` function.

```
FileMenuBar "Basic",columnmenu("Titles")+arraymenu(listchoices("Title",¶))
```

Multiple column menus are useful when your menus contain a lot of items.



On Windows computers menus always wrap to multiple columns, so this function is the same as the `menu()` function.

The WindowMenuBar Statement

The **WindowMenuBar** statement works exactly like the **FileMenuBar** statement except for the fact that it only specifies the menus for the current window instead of for all windows in the current database. Typically you would use this statement immediately after opening the window.

```
OpenForm "Currency Conversions"
WindowMenuBar "Basic",
    menu("Currency")+menuitems("Dollar;Euro;(Franc;(Lira;Peso;Pound;Yen"))
```

You can combine the **WindowMenuBar** and **FileMenuBar** statements. In this case the **WindowMenuBar** overrides for the windows in which it is used.

Advanced Topic: Live Menus Behind the Scenes

The following sections describes the nitty gritty details of how Live Menus work. For most applications you won't need to know this information and you can simply skip this section.

Live menus are set up by creating a fileglobal or windowglobal variable named **LiveMenuFormula**. This variable needs to be filled with a formula that calculates the menus to display. An important distinction is that the variable must contain with the formula itself -- not the result of the formula. The **FileMenuBar** and **WindowMenuBar** statements automatically create this variable for you and fill it with the formula.

When switching to a new window or clicking on the menu bar Panorama will look for this variable. If the variable exists, Panorama will calculate the formula result, and use that result to display all of the menus. This means that all of the menu contents are calculated on the fly as you click on the menu bar!

The formula in the **LiveMenuFormula** variable must calculate a carriage return delimited array. Panorama will scan this array from top to bottom. A line that begins with (and ends with) signals the start of a new menu. All other lines represent individual menu items.

Menu Titles

Menu titles must begin and end with (and). Here are some examples of menu titles.

```
(File)
(Message)
(Windows)
```

A menu title can have one or more options. Options appear after the end of the menu title. A tab character must appear before the first option, immediately after the). The options available are:

***S** - This is a submenu. It will not appear in the menu bar, but may be attached to other menus (see below).

***E** - This is the edit menu. When a text or word processing object is clicked on, this menu will be replaced by the correct editing menu for that object.

***M** - This is a multiple column menu. Instead of scrolling if there are too many items to fit on the screen, the menu will wrap to two or more columns.

Here are some examples of menu titles with options:

```
(Fields)-*S*M
(Recent)-*S
```

Note: In these examples, the `␣` character represents the tab character. The actual formula to create the titles above would be:

```
"(Fields)"+"␣"+"*S*M"
"(Recent)"+"␣"+"*S"
```

To simplify the examples we'll use `↵` to represent the tab character in the examples in the following sections.

If a menu title is entirely numeric, Panorama will insert the corresponding standard menu into the menu bar at that location. For example, `(1)` is the **Apple** menu, while `(27)` is the standard **File** menu for use with a form. Note: If the user selects an item in a standard menu, the `.CustomMenu` procedure is not triggered. Instead, Panorama simply performs the normal action for that menu item. Note 2: A standard menu should not have any menu items (see below) following it. The standard menu should be immediately followed by another menu title.

Note: Panorama normally includes the **Action** and **Wizard** menus in your menu bar automatically. If you want to remove either of these menus you can by using the special menu titles `(-ACTION)` and `(-WIZARDS)`.

Menu Items

Any line that does not begin with a `(` (and end with `)` is a menu item. Most menus have one or more items. The items should be immediately below the corresponding menu title. Here is an example of a simple menu:

```
(Shipping)
US Mail
UPS Ground
Fedex
DHL
```

If the first character of a menu item is `√` or `•` the menu will be marked:

```
(Shipping)
US Mail
√UPS Ground
Fedex
DHL
```

Note: In this example, the name of the menu item is still `"UPS Ground"`, not `"√UPS Ground"`. This is the value that will appear in `info("trigger")` if this menu item is selected.

If the first character of a menu item is `(` the menu will be disabled (dimmed). In this example both Fedex and DHL are not available:

```
(Shipping)
US Mail
√UPS Ground
(Fedex
(DHL
```

If the first character of a menu item is a tab then special characters in the name are put into the menu name. If the `LiveMenu` formula results in the text shown below, the `(Alpha` menu item will not be dimmed, and the `(` character will appear in the menu. The `(Beta)` menu item is NOT a new menu title, and the `(` and `)` will appear in the menu. (Remember that in these examples `↵` represents the tab character.

```
↵(Alpha
↵(Beta)
```

You can combine the `√`, `•`, `(` and `↵` characters. In this example the Gamma menu is dimmed, checked, and contains the `(` and `)` characters.

```
√(↵(Gamma)
```

By adding `↵` at the end of the menu item you can specify options

```
MENU ITEM↵/F - Command-F
MENU ITEM↵/N - Command-N
MENU ITEM↵<B - Bold
MENU ITEM↵<I - Italic
MENU ITEM↵<U - Underline
```

All of these options can be combined in ways that make sense. The Omega menu item below is checked with a bullet (`•`), is bold, italic and has an key equivalent of Command-F.

```
•↵(Omega)↵/F<B<I
```

Submenus

A submenu must be defined before it is used. Once it is defined it can be attached to any menu item like this:

```
MENU ITEM↵(SUBMENU)
```

Here is an example that shows how to create a Travel menu with three submenus, Air, Car, and Train

```
(Airlines)↵*S
American
Continental
Delta
Jet Blue
Southwest
United
(Train)↵*S
Amtrak
VIA
(Car Rental)↵*S
Avis
Hertz
National
(Travel)
Air↵(Airlines)
Car↵(Car Rental)
Train↵(Train)
```

Formula Errors

What if the formula in the [LiveMenuFormula](#) variable contains an error? In that case, Panorama simply displays the standard menus. There is no error message or indication of where the problem occurred. So if your custom menu doesn't appear you need to start looking very closely at the formula you have set up. (Note: Another advantage of the [WINDOWMENUBAR](#) and [FILEMENUBAR](#) statements is that they automatically check for and display formula syntax errors when you write your procedure. However, it is still possible to have a formula evaluation error (for example "number when text was expected") even when using these statements. Often the best approach is to simply remove parts of the formula until it works, then fix the portion that was most recently removed.)

The `menu()`, `menuitems()`, `arraymenu()` and `checkarraymenu()` functions

These functions were introduced earlier. Now you can easily see what these function actually do. The `menu()` function simply adds `()` and a carriage return to the menu name. The `menuitems()` function simply converts a semicolon separated array into a carriage return separated array. It also converts `↵` characters into tab characters. The `arraymenu()` functions add a tab to the beginning of each array element so that `(,), *, <` and `/` characters in array values will be displayed properly in the menu.

Helper Functions for Standard Menus

The following functions help incorporate standard Panorama menus into your live menu bar. By using these functions you eliminate the need to look up the correct menu numbers. In addition, the functions automatically adjust the menu numbers depending on whether or not the current window is a form or data sheet.

STANDARDFILEMENU() - Generates the specification for the standard File menu (and the Window sub-menu). The specification will be adjusted depending on whether the current window is a form or a data sheet.

STANDARDEDITMENU() - Generates the specification for the standard Edit menu. The specification will be adjusted depending on whether the current window is a form or a data sheet.

STANDARDVIEWMENU() - Generates the specification for the standard View menu.

STANDARDFIELDSMENU() - If the current window is a data sheet, this function generates the specification for the standard Fields menu.

STANDARDTEXTMENU() - If the current window is a data sheet, this function generates the specification for the standard Text menu (as well as the Font and Size menus).

STANDARDSEARCHMENU() - Generates the specification for the standard Search menu.

STANDARDSORTMENU() - Generates the specification for the standard Sort menu.

STANDARDMATHMENU() - Generates the specification for the standard Math menu.

STANDARDSETUPMENU() - Generates the specification for the standard Edit menu. The specification will be adjusted depending on whether the current window is a form or a data sheet.

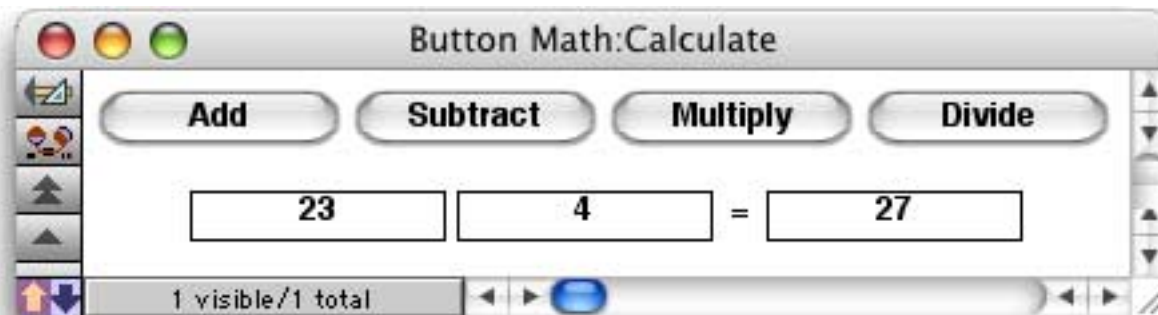
NOACTIONMENU() - When included in the formula the Action menu is removed from the menu bar.

NOWIZARDMENU() - When included in the formula the Wizards menu is removed from the menu bar.

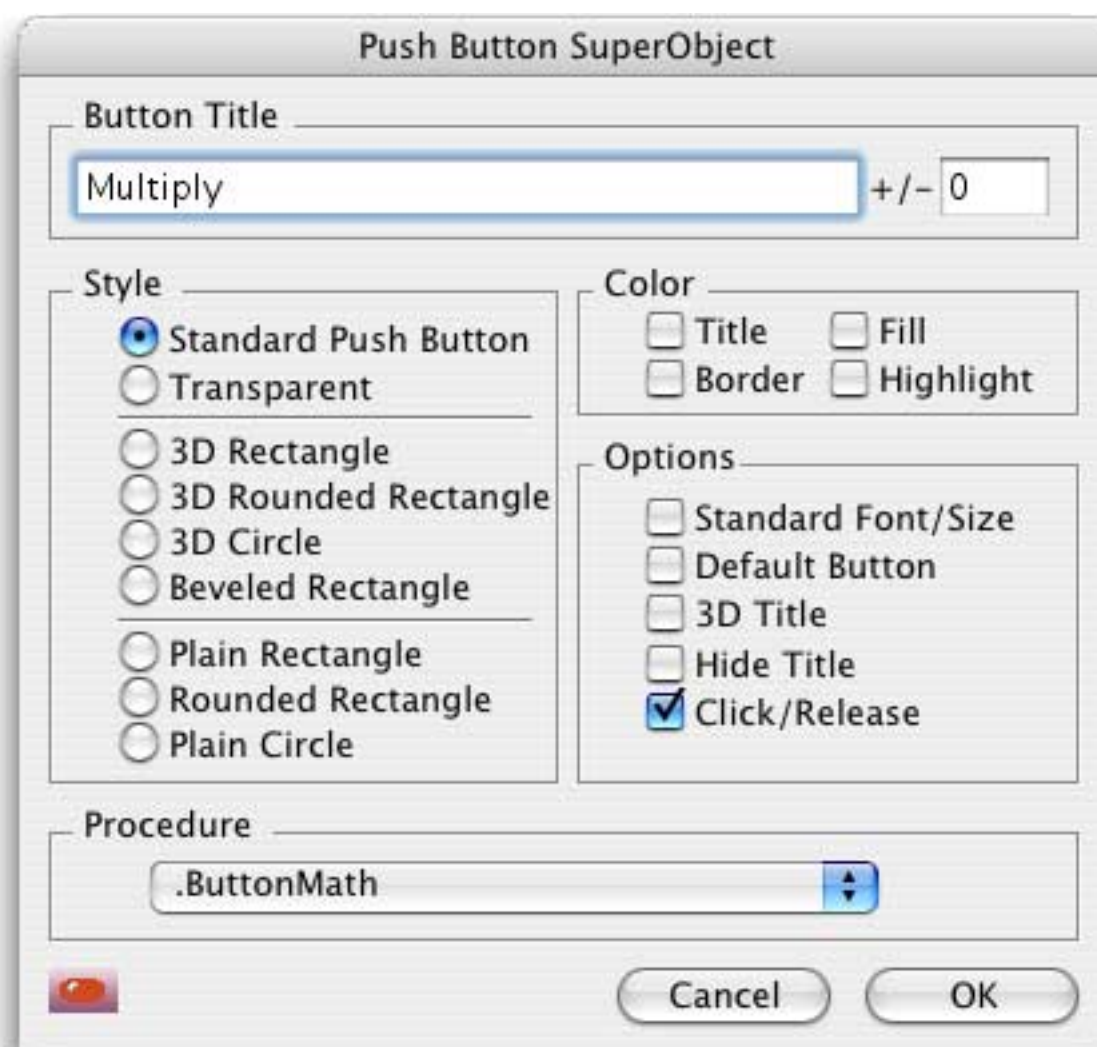
Buttons

Buttons are an important part of the today's modern graphic user interfaces. You can use a wide variety of buttons in any Panorama form. Panorama buttons come in three basic varieties: push buttons, data buttons (checkboxes and radio buttons), and pop-up menu buttons. All of these types of buttons can trigger a procedure. Use the configuration dialog for the button to select which button will be triggered when the button is pressed (see "[Buttons & Widgets](#)" on page 823 of the *Panorama Handbook*).

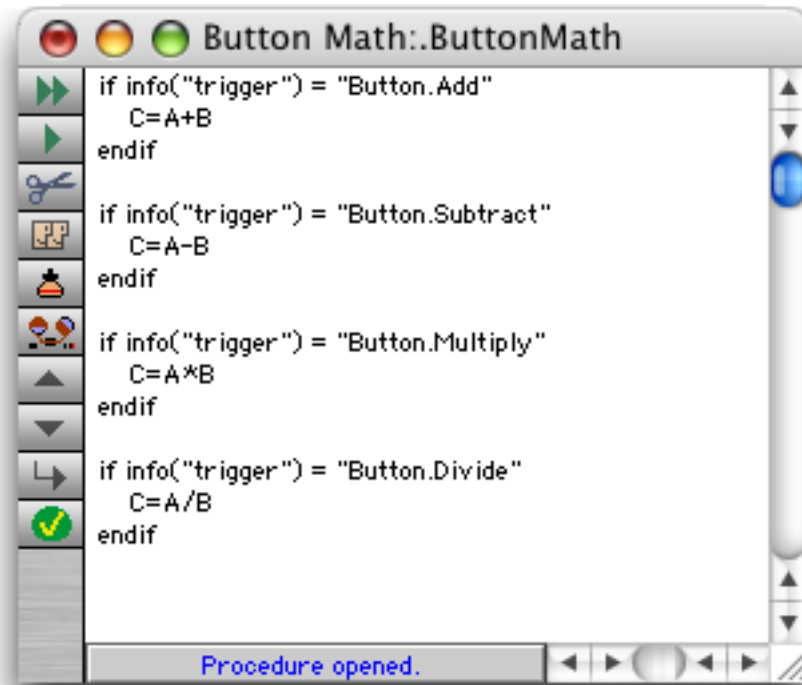
When a button is triggered by a procedure the `info("trigger")` function will return the title of the button. If you wish you may use a single procedure with many different buttons. For example, consider this form, which has four different buttons.



All four of these buttons trigger the `.ButtonMath` procedure. In fact, the only difference between these buttons is their titles. Here is the configuration dialog for one of these buttons.



The `.ButtonMath` procedure uses the `info("trigger")` function to decide which button was pressed.



Notice that the name of this procedure starts with a period. This makes this an “unlisted” procedure that does not appear in the **Action** menu (see See “[Unlisted Procedures](#)” on page 361). It wouldn’t make any sense to trigger this procedure from the menu, so it’s best to make it unlisted.

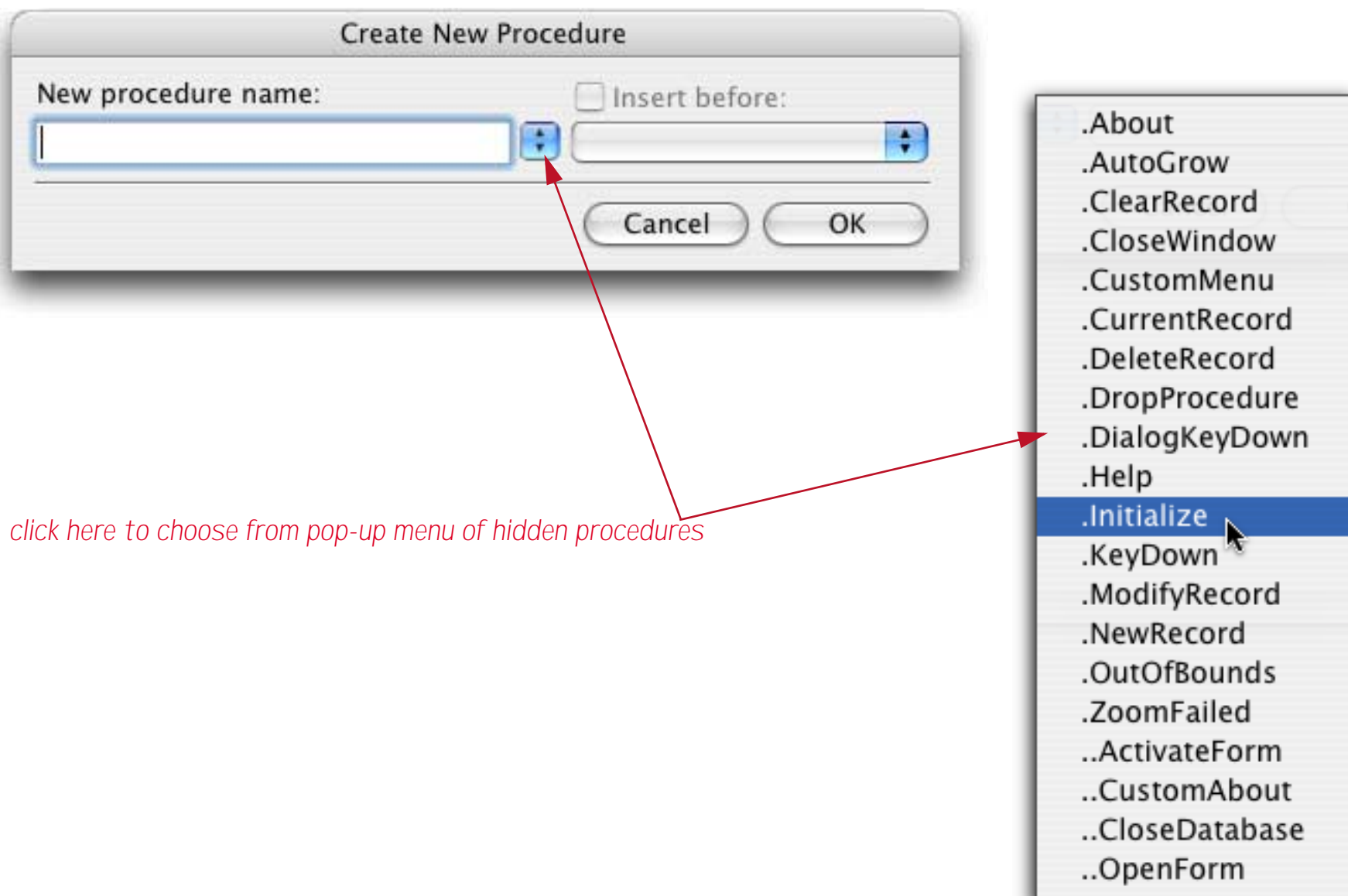
Hidden Triggers

Hidden triggers activate a procedure automatically when the user performs some normal Panorama action. Examples of user actions that can cause a hidden trigger to activate include adding new records to a database, deleting records, opening a file, closing a window and many more. The trigger is “hidden” because the user is not explicitly asking Panorama to activate a procedure by pressing a button or selecting a menu choice.

Procedures that are activated by hidden triggers can modify (or even override) the way Panorama reacts to many standard user actions. For example, when a user clicks on a window’s close box, Panorama normally responds by closing the window. But with a hidden trigger the programmer can activate a procedure whenever the close box is clicked. This procedure can do anything the programmer wants. For example, the programmer may want to save the window position before the window is closed. Or the programmer may not want to let the user even close the window until all the data on a form is filled in. Of course this kind of flexibility comes with a price. The user expects the window to close—so any other action must be carefully designed so that it doesn’t confuse or frustrate the user.

Creating Hidden Trigger Procedures

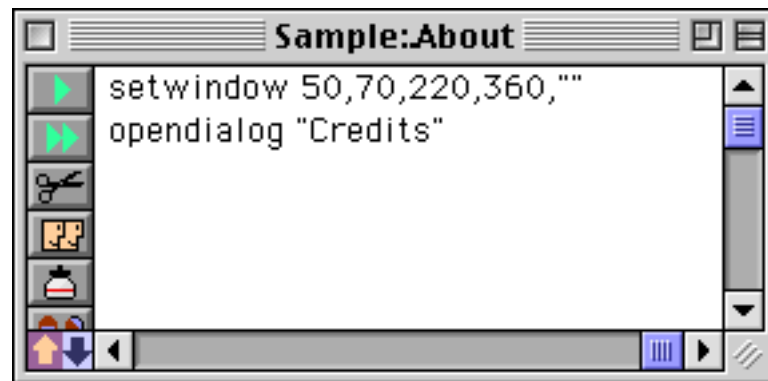
To create a procedure that is activated by a hidden trigger you must give the procedure a special name. For example, suppose you want to create a procedure that is triggered whenever a window is closed. That procedure must be named **.CloseWindow**. If a database contains a procedure with that name, it will always be triggered when the user clicks on the close box of a window in that database. To make it easier to create hidden trigger procedures, the **New Procedure** dialog contains a pop-up menu of the special procedure names required for hidden triggers.



Each of these possible hidden trigger procedures is explained in the following sections.

.About

This hidden trigger procedure will be triggered when the user selects **About Panorama** from the Apple menu. Normally selecting this menu item displays the “splash screen” and copyright message for Panorama. Using the **.About** hidden trigger procedure, you can display your own splash screen with information about your database. The following example shows a typical **.About** hidden trigger procedure that opens a form called Credits. This form would normally include a picture or logo for your database. It should also include a button that allows the user to close the window and resume normal use of the database (or you could even flip through multiple credit pages before closing the window).

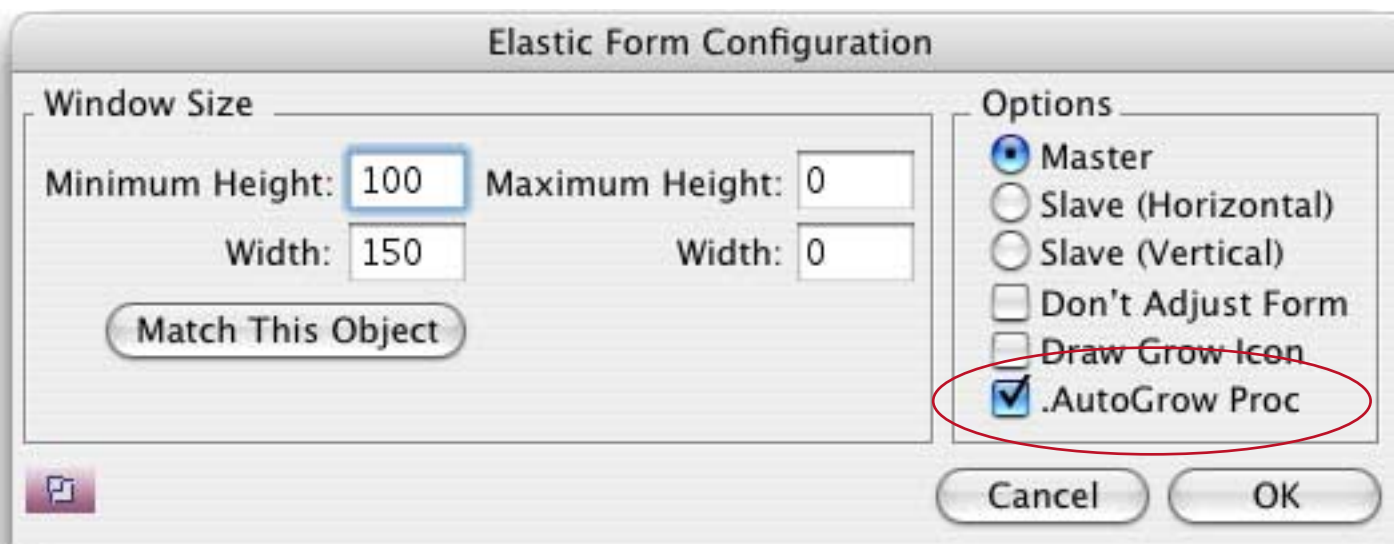


Warning: Your revised credit screen **must** include the ProVUE copyright notice in its entirety.

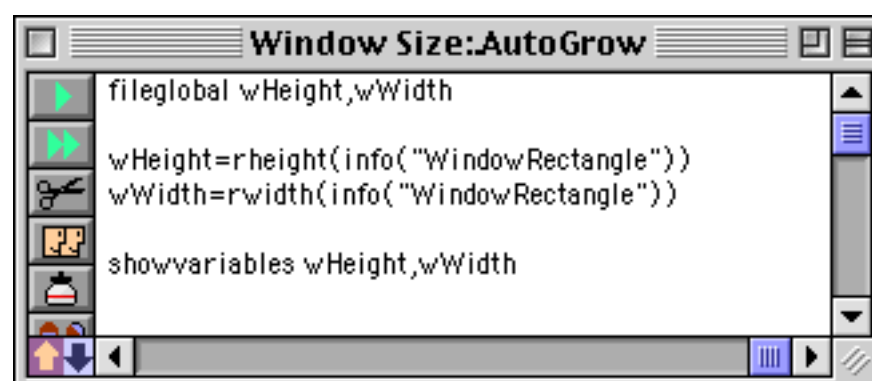
.Note: You can also change the name of the About Panorama menu item. See “[..CustomAbout](#)” on page 397.

AutoGrow

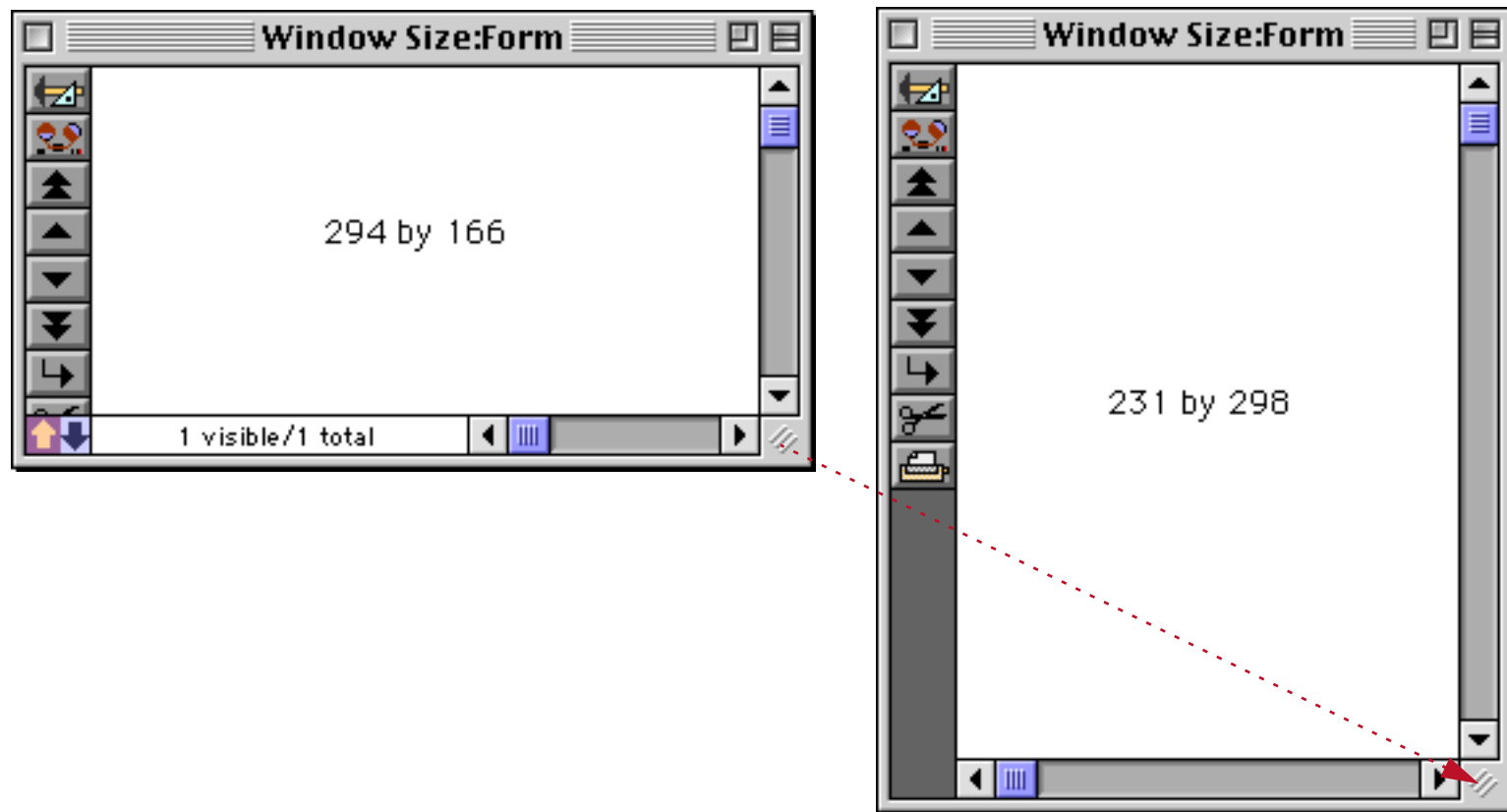
The **.AutoGrow** procedure is designed to work with Elastic forms (see “[Elastic Forms](#)” on page 922 of the *Panorama Handbook*). To use this procedure you must enabled the option in the auto-grow objects, like this.



When this option is enabled the **.AutoGrow** procedure will be triggered every time the window changes size. Here is an **.AutoGrow** procedure that simply calculates the width and height of the window.



In this database the **.AutoGrow** procedure is simply used to display the size of the new window.



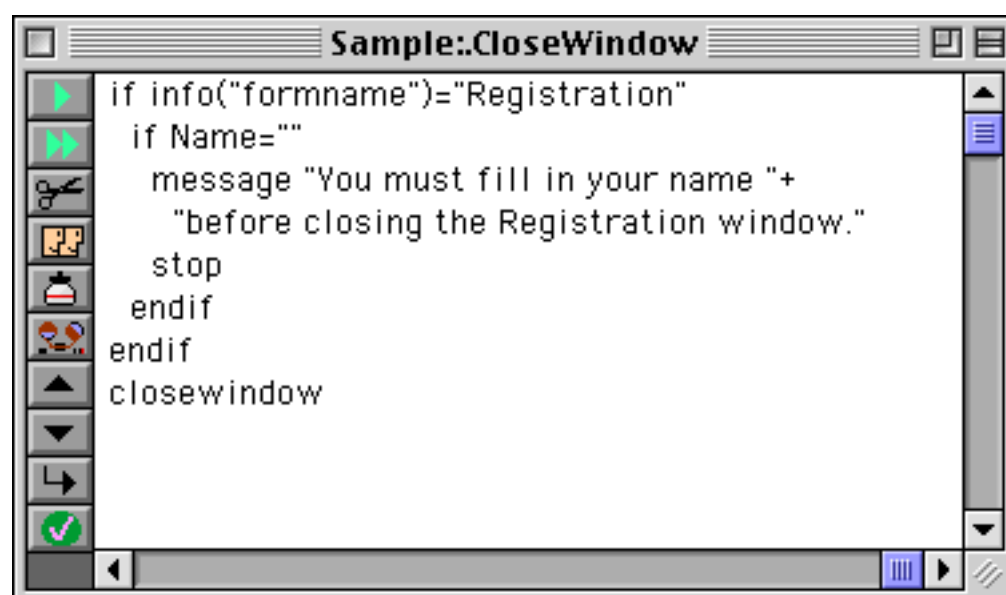
A more useful application would be to adjust elements of the form depending on the size of the window. See [“Programming Graphic Objects on the Fly”](#) on page 633 to learn how to adjust form elements.

.ClearRecord

This procedure is triggered when you choose the **Clear** menu item from the Edit menu, but only in a crosstab. Frankly, we can't remember why this feature was added!

.CloseWindow

This hidden trigger procedure will be triggered when the user clicks on the close box in the upper left hand corner of a window. Usually clicking on this box causes the window to close. Using the **.CloseWindow** hidden trigger procedure you can perform extra steps before closing the window, or even prevent the window from closing. Here is a typical example of how the **.CloseWindow** hidden trigger procedure is used. If the **Registration** form is open, the procedure checks to make sure that the **Name** field is not empty. If the **Name** field is empty, the procedure tells the user that they cannot close the form yet. Otherwise the procedure goes ahead and closes the window.



Notice that the last statement in this procedure, `closewindow`, actually closes the window. It does not trigger the procedure again. Only a user action, such as clicking or pressing a key, can trigger a hidden procedure.

Warning: The `.CloseWindow` procedure is triggered only by the user clicking on the close box of the window. It is not triggered by other actions that might close the window, such as closing the entire file or quitting from Panorama.

.CurrentRecord

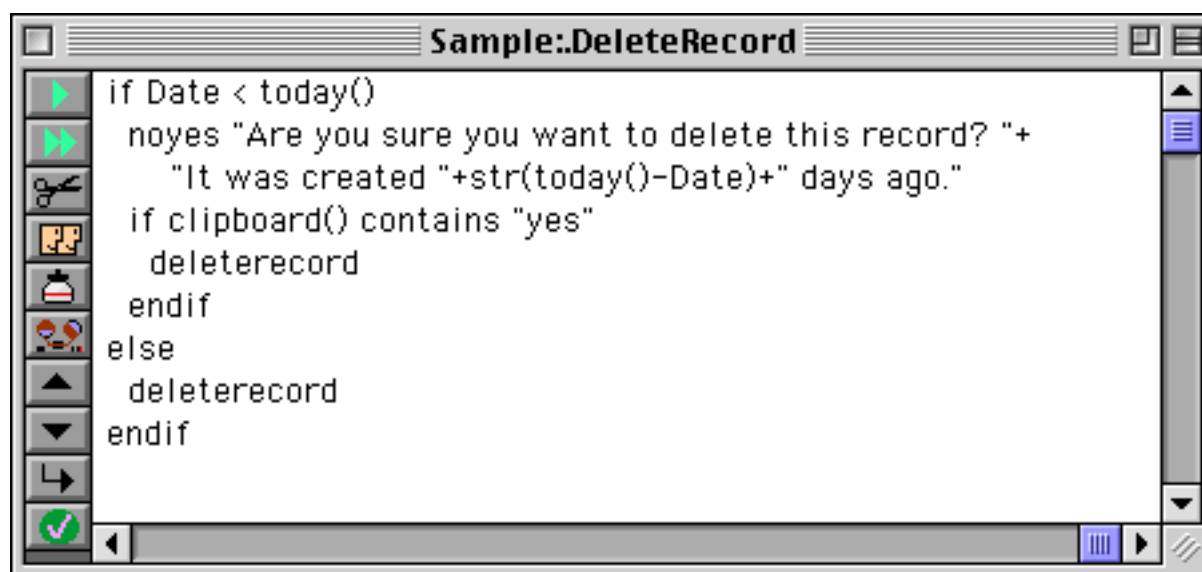
This hidden trigger procedure will be triggered when the database shifts to a different record. For example, this procedure is triggered when you move up or down in the database with the vertical scroll bar, or with the **Find** or **Find Next** commands. (Remember, like other hidden trigger procedures it is not triggered by procedure statements, only by user actions.) You can use this procedure to perform any special actions that are necessary to display or work with this record.

.CustomMenu

This hidden trigger procedure will be triggered when the user selects a Custom menu item. For a complete description of custom menus and the `.CustomMenu` hidden trigger procedure see "[The .CustomMenu Procedure](#)" on page 363.

.DeleteRecord

This hidden trigger procedure will be triggered when the user attempts to delete a record from the database. This procedure could be triggered by the **Delete Record** tool, or by pressing the **Delete** key in data sheet or view-as-list windows. The example below allows records created today to be deleted immediately, but double checks before allowing older records to be deleted.



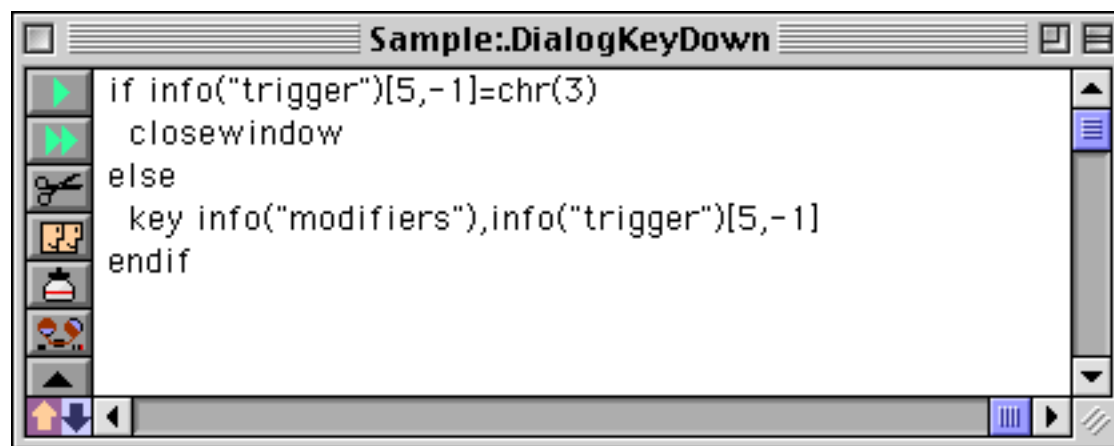
Notice that the record is not deleted unless the procedure deletes the record. The `.DeleteRecord` procedure interrupts the normal deletion process and takes over. This puts you, the programmer, in control.

.DialogKeyDown

This very specialized hidden trigger procedure will be triggered when the user presses a key in a form that has no drag bar (a form that looks like a dialog). However, this procedure will not be triggered if a Data Cell or SuperObject Text Editor is currently active. So if you are in a dialog created with a Panorama form, not editing text, and press a key, the `.DialogKeyDown` procedure will be triggered. (Another way to intercept keystrokes is with a hotkey procedure, see "[Hot Key Procedures](#)" on page 390.)

The procedure can tell what key was pressed by using the `info("trigger")` function. For example, if the user presses **Y** the `info("trigger")` functions will return `Key.Y`.

Here is a [.DialogKeyDown](#) procedure that closes the dialog window if the user presses the **Enter** key. All other keystrokes will be processed normally.



```

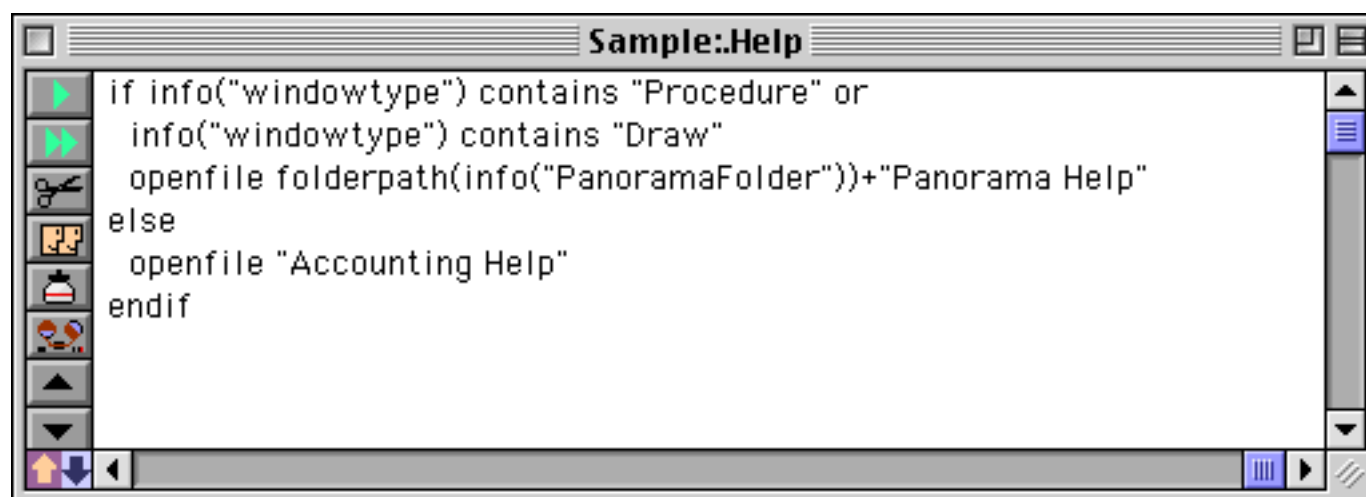
if info("trigger")[5,-1]=chr(3)
  closewindow
else
  key info("modifiers"),info("trigger")[5,-1]
endif

```

To process keystrokes normally this procedure uses the key statement. See “[.KeyDown](#)” on page 382 for more information on this statement.

.Help

This hidden trigger procedure will be triggered when the user selects **Help** from the Apple menu. Normally selecting this menu item opens the Panorama help system. Using this hidden trigger procedure you can force Panorama to use your own custom help system for your database. The example shown below will open the normal Panorama Help if the current window is a procedure or a form in graphics mode. Otherwise it will open the special [Accounting Help](#) database.



```

if info("windowtype") contains "Procedure" or
  info("windowtype") contains "Draw"
  openfile folderpath(info("PanoramaFolder"))+"Panorama Help"
else
  openfile "Accounting Help"
endif

```

.Initialize



This hidden trigger procedure will be triggered when the database is opened, either from the desktop or from the **Open** dialog. You can use this procedure to initialize global and permanent variables, open resource files, set up custom menus, pre-sort or pre-select the database...anything that needs to be done automatically whenever the database file is opened.

Warning: Most procedures can only be triggered from the data sheet or a form. However, the [.Initialize](#) procedure will start running immediately when the file is opened, in whatever window happens to be open. If this window is not a data sheet or form, the procedure may not operate correctly. Many procedure statements (sort, group, select, etc.) will not operate properly from a non-data window. If this may happen, the first thing the [.Initialize](#) procedure should do is open a form or the data sheet.

.KeyDown

This very specialized hidden trigger procedure will be triggered when the user presses a key in a form. However, this procedure will not be triggered if a Data Cell or SuperObject Text Editor is currently active. So if you are in a form, not editing text, and press a key, the [.KeyDown](#) procedure will be triggered. (Another way to intercept keystrokes is with a hotkey procedure, see “[Hot Key Procedures](#)” on page 390.)

The procedure can tell what key was pressed by using the `info("trigger")` function. For example, if the user presses Y the `info("trigger")` functions will return Key.Y. Special keys will return the special values listed in this table:

Tab		chr(9)
Enter		chr(3)
Return		chr(13)
Esc		chr(27)
Delete		chr(8)
Left Arrow		chr(28)
Right Arrow		chr(29)
Up Arrow		chr(31)
Down Arrow		chr(30)

The procedure can use the `info("modifier")` function to tell what modifier keys have been pressed along with the primary key: shift, option, control, and command.

Here is a **.KeyDown** procedure that closes the window if the user presses the **Enter** key. If the user presses the **\$** key Panorama jumps to the **Amount** data cell and begins to edit it. All other keystrokes will be processed normally.

```
local KeyStroke
KeyStroke=info("trigger")[5,-1]
case KeyStroke=chr(3)/* enter key */
  closewindow
case KeyStroke="$"
  field Amount
  editcellstop
defaultcase
  key info("modifiers"),KeyStroke
endcase
```

To process keystrokes normally this procedure uses the `key` statement. The `key` statement passes the key-stroke back to Panorama for normal processing (see “**KEY**” on page 5461 of the *Panorama Reference*). This statement has two parameters: 1) the modifiers associated with the key, and 2) the key itself.

.ModifyRecord

This hidden trigger procedure will be triggered when the user modifies any field in the database. Warning: The **.ModifyRecord** procedure will not run if a procedure is already running, or if the field has its own procedure. In those cases you may want the other procedure to call the **.ModifyRecord** procedure as a subroutine (more on this in a moment). The **.ModifyRecord** procedure is also not called if the data is modified with a command in the **Fill** menu, see **.ModifyFill** below.

The **.ModifyRecord** procedure example below automatically marks the latest date and time when a record was modified. This example assumes that the database has two fields for time/date tracking: **ModifyDate** (a date field) and **ModifyTime** (a numeric field).

```
ModifyDate=today()
ModifyTime=now()
```

Note: This example illustrates the **.ModifyRecord** procedure, but a better way to perform this task would be to create a Time Stamp field in the design sheet. See “[Automatic Time/Date Stamping](#)” on page 301 of the *Panorama Handbook* for details on this process.

If your database has other procedures that modify the database they should call the **.ModifyRecord** procedure to make sure that the time stamp is kept up to date. For example, here is a procedure that automatically subtracts one from the **QtyInStock** field.

```
QtyInStock=QtyInStock-1
call .ModifyRecord
```

.ModifyFill

This hidden trigger procedure will run when the user uses most commands in the Math menu (**Fill**, **Formula Fill**, etc.). The **.ModifyFill** procedure will not run if a procedure is already running, or if the field has its own procedure. In those cases you may want the other procedure to call the **.ModifyFill** procedure as a subroutine (more on this in a moment).

The **.ModifyFill** procedure example below automatically marks the latest date and time when a fill command is used. This example assumes that the database has two fields for time/date tracking: **ModifyDate** (a date field) and **ModifyTime** (a numeric field).

```
field ModifyDate
formulafill today()
field ModifyTime
formulafill now()
```

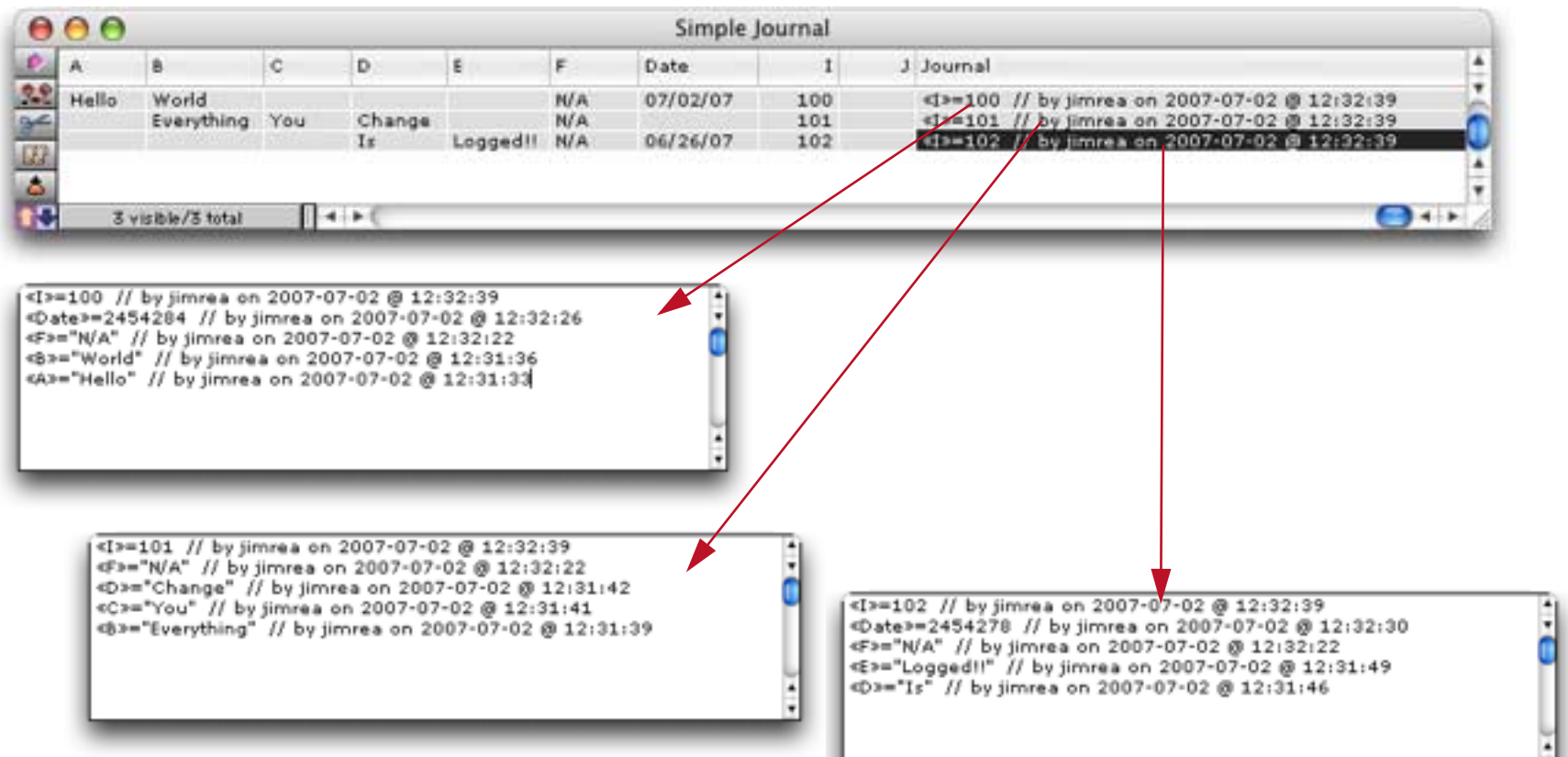
If your database has other procedures that use fill statements they should explicitly call the **.ModifyFill** procedure to make sure that the time stamp is kept up to date. The example below selects all items that have over 500 in stock and have not been touched in 30 days. For those items, it reduces the price by 10%, then marks the modification date and time.

```
select QtyInStock>500 and today()-ModifyDate>30
field Price
formulafill Price*0.90
call .ModifyFill
```

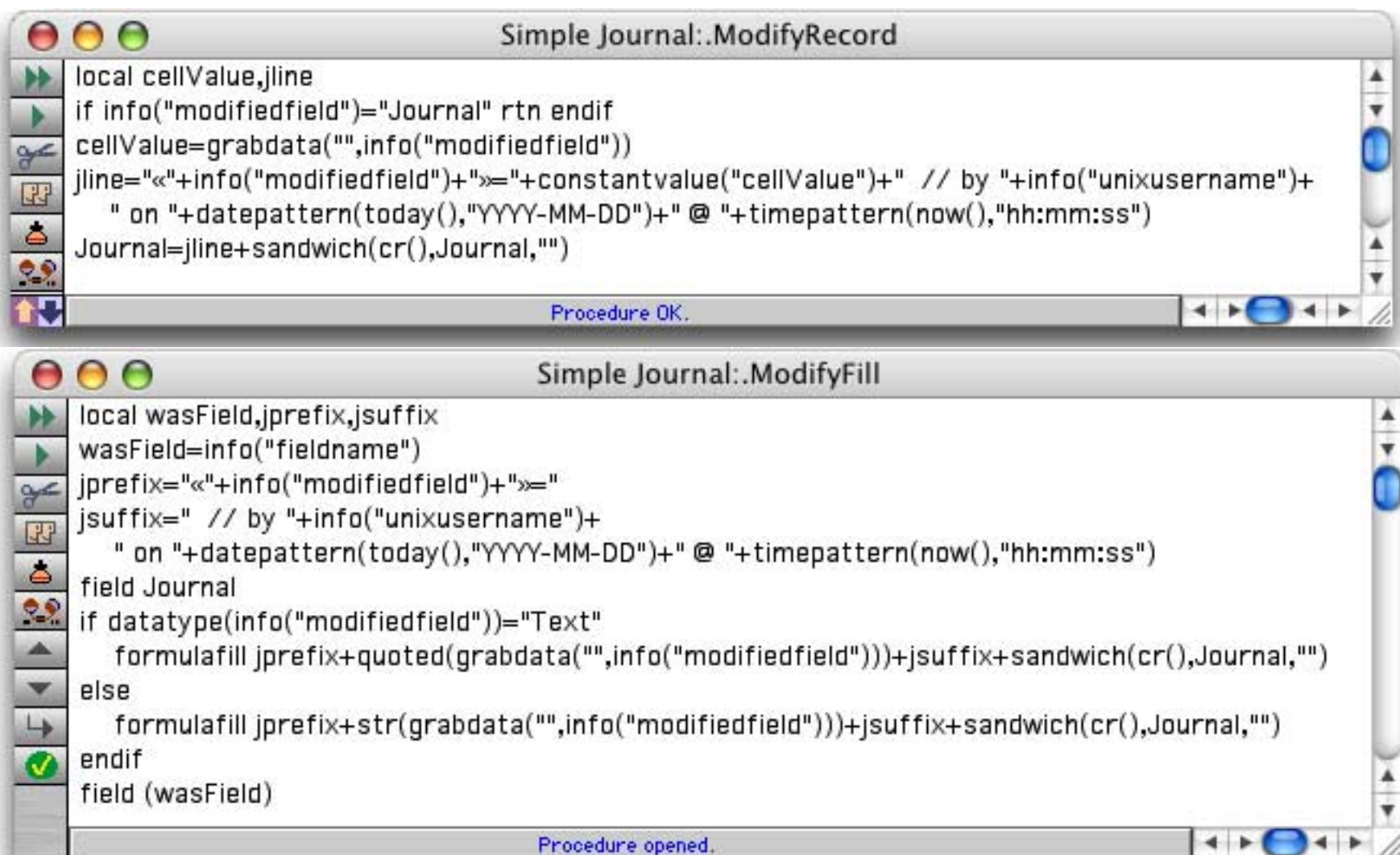
Since the **.ModifyFill** procedure is not triggered by a **FormulaFill** statement in a procedure, the procedure must update the modification date and time itself by explicitly calling **.ModifyFill**.

Logging Changes (Audit Trail) with .ModifyRecord, .ModifyFill and info("modifiedfield")

It's possible to create a log of all changes made to a database, so you can see who changed what when. An example database included with Panorama illustrates this. The example database is called **Simple Journal**.



As the illustration above shows the **Journal** field contains a log of all changes made to the database. This log is created by the **.ModifyRecord** and **.ModifyFill** procedures in the database.



One possible problem with this logging mechanism is that the log quickly becomes larger than the actual data! We're sure, however, that some of you will find this a valuable tool.

.NewRecord

This hidden trigger procedure will be triggered when the user attempts to add a new record to the database with the **Add New Record** or **Insert New Record** tool, or by pressing the **Return** key in the data sheet or a view-as-list form (see “[Adding a New Record](#)” on page 268 of the *Panorama Handbook*). The `info("trigger")` function can be used to determine which of the three possible actions triggered the procedure:

"New.Add"	Add New Record tool or Add New Record menu item
"New.Insert"	Insert New Record tool
"New.Return"	Return key

This sample **.NewRecord** procedure won't allow new records to be added if there already at 100 or more records in the database:

```
if info("records")>=100
  message "Sorry, this database is limited to 100 records."
else
  case info("trigger") = "New.Add"
    addrecord
  case info("trigger") = "New.Insert"
    insertrecord
  case info("trigger") = "New.Return"
    insertbelow
  endcase
endif
```

Here's another **.NewRecord** example that only allows new records to be added to the end of the database, not inserted in the middle. In this example the case for `info("trigger") = "New.Insert"` has been eliminated, because we know that a new record cannot be inserted. The case for `info("trigger") = "New.Return"` has been expanded to also check to see if we are on the last line of the database with the `info("eof")` function.

```
local AddFlag
AddFlag="no"
case info("trigger") = "New.Add"
  addrecord
  AddFlag="yes"
case info("trigger") = "New.Return" and info("eof")
  insertbelow
  AddFlag="yes"
endcase
if AddFlag="no"
  message "Sorry, new records must be added"+
    " to the end of the database, not inserted in the middle."
  stop
else
  call .ModifyRecord
endif
```

At the end of this example procedure a message is displayed if the record could not be added. If the new record has been added this procedure calls **.ModifyRecord** (see “[.ModifyRecord](#)” on page 383). The procedure could also calculate default values at this point.

Warning: The **.NewRecord** procedure is not triggered when new records are added by appending (with the **Open File** command).

.OutOfBounds

This very specialized hidden trigger procedure will be triggered when a form that has no drag bar is open (a form that looks like a dialog) and the user clicks outside of the window. In other words, if you create a dialog with a Panorama form and the user clicks outside of the dialog, this procedure will be triggered. (If there is no **.OutOfBounds** procedure, Panorama will simply beep when this happens.)

If you wish, the **.OutOfBounds** procedure could simply close the window if the user clicks outside of it:

```
closewindow
```

Or, the **.OutOfBounds** procedure could display a message:

```
message "Please click inside the dialog."
```

Of course, this message might make the user feel like they were back in kindergarten (please draw inside the lines!).

.ZoomFailed

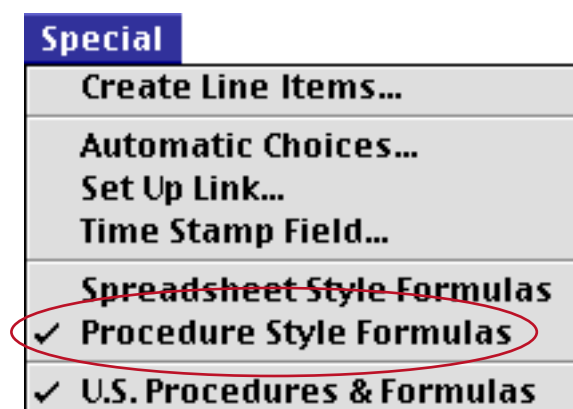
This very specialized hidden trigger procedure will be triggered when the user clicks on the zoom box, but Panorama cannot zoom the window. The only time this happens is if both the horizontal and vertical scroll bars are disabled. In that case you can use the **.ZoomFailed** procedure to make the window zoom, perhaps by switching to a different form as the example below shows:

```
case info("formname")="Letter
  goform "Full Letter"
  setwindow 20,2,760,500,"
  zoomwindow
case info("formname")="Full Letter"
  goform "Letter"
  setwindow 20,2,320,500,"
  zoomwindow
endcase
```

Data Entry Triggers

Panorama can automatically trigger a procedure whenever you enter new data into a database field. Unlike other hidden trigger procedures, data entry trigger procedures do not have a special name. Instead, the name of the procedure is specified in the design sheet.

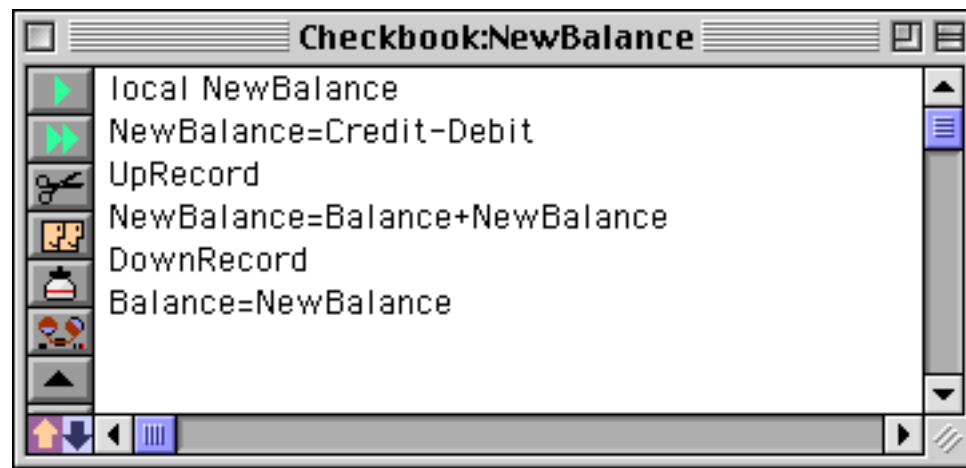
To set up a procedure that is triggered by data entry, first create the procedure. If you don't want the procedure to be listed in the **Action** menu the procedure name should start with a period. The procedure name must be a single word with no spaces. Then open the design sheet and make sure that the **Procedure Style Formulas** option is checked in the **Special** menu. If you have been using **Spreadsheet Style** formulas you will need to adjust the other formulas that have been set up in the design sheet (see "[Automatic Calculations](#)" on page 303 of the *Panorama Handbook*).



Then type the procedure name into the **Equation** column for the field. If there are already formulas for the field, the procedure name should be typed after the last formula.

Once the data entry trigger is set up, the procedure will be triggered automatically each time the user presses **Enter**, **Return** or **Tab** to enter data. The procedure can find out which key was pressed using the `info("trigger")` function. If the **Return** key was pressed, `info("trigger")` will be `Key.Return`. If the **Tab** key was pressed, `info("trigger")` will be `Key.Tab`.

Here is a sample data entry triggered procedure that calculates the new balance for a checkbook. The data sheet should be set up so that this procedure is triggered whenever the **Credit** or **Debit** fields are modified.



To set this procedure up to be triggered you must enter it into the Equation field of the design sheet, like this.

Field Name	Type	Di	Align	Out	Inp	Range	Choi	Link	Clair	Tab	Cap	Dup	Def	Equation	Reac	Writ	Wik
↓Date	Date	0	Left			Any		Off	Off	Off	Yes	tod.			0	0	8
↓CkNum	Num	0	Right			Any		Off	Off	Off	Yes	+			0	0	4
↓PayTo	Text	0	Left			Any		On	Off	Wor	Yes				0	0	16
↓Category	Text	0	Left			Any		On	Off	Wor	Yes				0	0	12
↓Debit	Num	2	Right	#.		Any		Off	Off	Off	Yes		NewBalance		0	0	10
↓Credit	Num	2	Right	#.		Any		Off	Off	Off	Yes		NewBalance		0	0	9
↓Balance	Num	2	Right	#.		Any		Off	Off	Off	Yes				0	0	9
↓Memo	Text	0	Left			Any		Off	Off	Off	Yes				0	0	22

When the user presses the **Tab** key while editing data, Panorama normally skips to the next field. However, if the procedure uses the field statement to switch to a different field, Panorama's normal tab order is aborted. If you want to abort the tab order without moving to a different field, use the `stoptab` statement. Here is a sample data entry triggered procedure that filters out negative values in the **Price** field.

```
if Price < 0
  message "Negative prices are not allowed"
  stoptab
endif
```

If a data entry triggered procedure is triggered, the **.ModifyRecord** procedure (if any) for the database will not be triggered. If necessary, you should call to **.ModifyRecord** somewhere in your data entry triggered procedure. Here's a revised version of our check balance procedure.

```
local NewBalance
NewBalance=Credit-Debit
UpRecord
NewBalance=Balance+NewBalance
DownRecord
Balance=NewBalance
call .ModifyRecord
```

Another option is to get rid of the data entry triggered procedures completely and do all the work in the **.ModifyRecord** procedure. That option is described in the next section.

Data Entry Triggers (Part Two)

Instead of using separate data entry trigger procedures for each field as described in the last section, you combine all of the data entry procedures for the entire database into a single procedure: the **.ModifyRecord** procedure. Remember, this procedure is triggered whenever any field is modified. This procedure can use the **info("fieldname")** function to determine which field was modified, and take appropriate action. Here's an example of a **.ModifyRecord** procedure that performs special actions for the **Date**, **Credit**, and **Debit** fields in a checkbook database.










```
case info("fieldname")="Date"
  if Date>today()
    Date=today()
    stoptab
    message "Post-dated checks not allowed!"
  endif
case info("fieldname")="Credit" or info("fieldname")="Debit"
  local NewBalance
  NewBalance=Credit-Debit
  UpRecord
  NewBalance=Balance+NewBalance
  DownRecord
  Balance=NewBalance
endcase
ModifyDate=today()
```

This procedure also time stamps the **ModifyDate** field in the current record whenever any field is modified. With this technique it's all in one procedure and easy to keep track of. Another advantage of this technique is that it will work just fine with the **Spreadsheet Style Formulas** option.

Hot Key Procedures

Panorama can be configured so that an action is triggered automatically when a particular key or key combination is pressed. Unless you want a special effect (for example a hotkey that only works in a single window) the easiest way to set that up is to use the Hotkey wizard, which is in the Preferences submenu of the Wizard menu. Using this wizard you can set up a hotkey (or multiple hotkeys) that triggers a procedure in your database. The wizard automatically manages the necessary permanent variables for you. See “[Hotkey Manager](#)” on page 99.

If you do need a special effect that is not supported by the Hotkey wizard you can set up hotkeys in a procedure. Every time a key is pressed Panorama looks for a special variable. This variable is named `HotKey[xx]`, where `xx` is the hexadecimal value of the keycode for the key that is pressed (see table below). For example, to create a hotkey procedure for the **F1** key the variable must be named `HotKey[7A]`. If Panorama finds such a value it takes the contents of the variable and executes them, just as if the variable was part of an `execute` statement (see “[Building Subroutines On The Fly \(The Execute Statement\)](#)” on page 280). The table below lists the codes for 96 different keys (the keys in green are on the numeric keypad).

A	00	M	2E	Y	10		37	0	1D		54		43	F9	65
B	0B	N	2D	Z	06		31	-	1B		55		4E	F10	6D
C	08	O	1F	[21	\	2A	=	18		56		45	F11	67
D	02	P	23]	1E	1	12		33		57	ESC	35	F12	6F
E	0E	Q	0C	;	29	2	13		32		58	F1	7A	F13	69
F	03	R	0F	'	27	3	14		24		59	F2	78	F14	6B
G	05	S	01	,	2B	4	15		7E		5B	F3	63	F15	71
H	04	T	11	.	2F	5	17		7D		5C	F4	76		73
I	22	U	20	/	2C	6	16		7B		4C	F5	60		74
J	26	V	09		38	7	1A		7C		47	F6	61		79
K	28	W	0D		3B	8	1C		52		51	F7	62	END	77
L	25	X	07		3A	9	19		53		4B	F8	64	HELP	72

The procedure below sets up a hotkey procedure for the **F1** key. After this procedure has been used pressing the **F1** key will open the [Favorite Databases](#) wizard.

```
global «HotKey[7A]»
«HotKey[7A]»={
  openfile folderpath(info("panoramafolder"))+"Wizards:Favorite Databases"
}
```

Because this hotkey procedure was set up as a global variable it will be active no matter what database is currently open. If you want to restrict the hotkey procedure to a particular database you should use a fileglobal variable. If you want to restrict the hotkey procedure to a particular window you should use a windowglobal variable.

If you define a hotkey procedure with a global or fileglobal variable keep in mind that the procedure may be triggered at **any time in any window**, so you must not make any assumptions about what window or even what type of window will be active when the procedure is triggered. It could be a procedure window, crosstab window, input window, you name it. Also, if you used a global variable you cannot even assume that the original database is still open.

Hotkey procedures do not work in Panorama dialogs. For example, pressing a hot key has no effect when you are using the **Find/Select** dialog, or the **Print** dialog, or any other dialog that is built into Panorama. However, hot keys do work in dialogs that you have created with a form (see “[Custom Dialogs](#)” on page 489).

To disable a hotkey procedure you can either set the hotkey variable to "" or you can destroy the variable with the undefine statement (see "[Destroying a Variable](#)" on page 249).

HotKeys with Modifiers

If you want to assign an action to a key with a specific modifier key combination, create a variable named `HotKey[kkmm]`, where *kk* is the hexadecimal value of the keycode for the key that is pressed (see table above) and *mm* is sum of the hexadecimal values for each modifier key. The table below lists the hex values for the four modifiers keys. (Please keep in mind that these are hexadecimal, not decimal values. You can use the RPN Programmer's Calculator wizard to calculate the sum of two or more hexadecimal numbers.)

Hex	Macintosh	Windows
01	Command	Control
02	Shift	Shift
08	Option	Alt
10	Control	Right Click

For example, to create a hotkey procedure for **Shift-F1** key the variable must be named `HotKey[7A02]`. To create a hotkey procedure for **Command-Option-A** the variable must be named `HotKey[7A09]`.

Universal HotKey Procedure

If Panorama does not find a hotkey for the specific key that has been pressed it will check to see if there is a variable named `HotKey[*]`. If there is, Panorama will execute the code it finds inside. Be very careful with this variable. For example, the procedure below will completely disable the keyboard until you **Quit** from Panorama. No matter what database you are in pressing the keyboard will no longer have any effect. Usually you will want to selectively apply a universal hotkey with a fileglobal or windowglobal variable.

```
global «HotKey[*]»
«HotKey[*]»="rtn"
```

The `.KeyDown` and `.DialogKeyDown` procedures provide another method to intercept keystrokes and process them yourself. See "[.KeyDown](#)" on page 382 and "[.DialogKeyDown](#)" on page 381.

Triggering a Procedure Every Second

Once every second Panorama checks for a special variable named `ExecuteEverySecond`. If it finds this variable, Panorama takes its contents and executes them, just as if the variable was part of an `execute` statement (see "[Building Subroutines On The Fly \(The Execute Statement\)](#)" on page 280). The most common use for this feature is to create animation within a form. For example, you could use this feature to make an item on the form blink, or you could use it to update a stopwatch display every second.

The `ExecuteEverySecond` variable may be a global, fileglobal, or windowglobal variable. If it is a `windowglobal` variable, the procedure will only be executed when that window is the front window. If it is a `fileglobal` variable the procedure will only be executed when that file is the active file (one of its windows is on top). If it is a `global` variable the procedure will be executed no matter what database is active, even if the original database that created the variable is no longer open. It's possible to have more than one `ExecuteEverySecond` variable active at a time, for example a windowglobal and a global. In that case Panorama will execute both of the procedures every second.

Here is an example that will cause a bullet to blink on and off once per second. The form **Blink Demo** should contain a Text Display SuperObject that displays the `blinkValue` variable, or a Flash Art object that uses this value as part of the formula. The `nowatchcursor` statement makes sure that the mouse arrow doesn't flip into a watch once per second as the procedure runs (see "[Disabling the Watch Cursor](#)" on page 310).

```
openform "Blink Demo"
windowglobal ExecuteEverySecond,blinkValue
blinkValue="•"
ExecuteEverySecond={
  nowatchcursor
  if blinkValue="•"
    blinkValue=""
  else
    blinkValue="•"
  endif
  showvariables blinkValue
}
```

Because the `ExecuteEverySecond` variable was defined as a windowglobal variable the object will only blink when that window is the front window. When another window comes to the front the blinking will pause. It will resume when the **Blink Demo** window is brought to the front again.

Since this example uses a windowglobal variable the procedure can assume that the **Blink Demo** window is on top when the procedure executes each second. If you use a fileglobal or global variable there is no guarantee what window will be on top. If you need to access a specific database you should use the secret window feature to temporarily activate it during the procedure (see "[Temporary "Invisible" Windows](#)" on page 454).

If you use a global variable you should be careful to be a good neighbor, since other databases may also be using the same variable. Instead of simply assigning the text of your procedure to the `ExecuteEverySecond` procedure you should append it. When you are done you should remove your code while leaving any other code. In addition, your procedure should not assume that the original database that activated it is still open — it may have been closed.

Panorama includes a Stopwatch wizard that can be used as a timer.



The three buttons in this form are tied to a single procedure which is listed below. When the **Start** button is pressed the procedure appends the code in `StopwatchCode` to the `ExecuteEverySecond` variable. This code will execute every second, causing the form to update as the timer runs. (Notice that the procedure also checks to make sure that the original database is still open using the `arraysearch()` and `info("files")` functions.) When the **Stop** button is pressed the procedure uses the `replace()` function to remove the code that it added without touching any other code that may have been added by other databases. In fact, you can make copies of this Stopwatch database and run them all at the same time. Each will keep its own time, and they will all keep running no matter what database is currently active.

```
global ExecuteEverySecond
fileglobal ElapsedTime,CumulativeTime,StartTime
local StopwatchCode
define ElapsedTime,0
define ExecuteEverySecond,""

StopwatchCode=
{ /* }+info("databasename")+{ Wizard */
nowatchcursor
if arraysearch(info("files"),)+{"+info("databasename")+"}"+{,1,¶}<>0
  local wasWindow
  wasWindow=info("windowname")
```



```

    window }+"{"+info("databasename")+"}"+"{:SECRET"
    ElapsedTime=CumulativeTime+now()-StartTime
    showvariables ElapsedTime
    window wasWindow
endif
}

if info("trigger") contains "Reset"
    ElapsedTime=0
    showvariables ElapsedTime
    rtn
endif
if info("trigger") contains "Start"
    if ExecuteEverySecond notcontains StopwatchCode
        CumulativeTime=ElapsedTime
        StartTime=now()
        ExecuteEverySecond=ExecuteEverySecond+StopwatchCode
    endif
endif
if info("trigger") contains "Stop"
    ExecuteEverySecond=replace(ExecuteEverySecond,StopwatchCode,"")
endif

```

If the code you assign to the [ExecuteEverySecond](#) variable contains an error Panorama will display an alert with the error message. It will then wait 20 seconds before it tries to execute the variable again. This delay gives you a chance to do something (perhaps simply quitting Panorama) before the error occurs again.

By the way, Panorama is not guaranteed to execute the procedure in the [ExecuteEverySecond](#) variable every single second. The procedure will not be executed when you are editing data, graphics, or a procedure. The procedure will not be executed when you hold down the mouse for an extended time, or if an operation like sorting, selecting or opening a file takes more than a second. Also, the procedure will not be executed if Panorama is not the frontmost program (in other words, if another program is on top).

Triggering a Procedure Every Minute

Once every minute Panorama checks for a special variable named [ExecuteEveryMinute](#) and executes the contents, if any. This variable is just like [ExecuteEverySecond](#) except that it only runs once per minute. At that rate this variable isn't much use for animations, but it can be used to check for reminders. This example beeps and displays the time once per hour.

```

fileglobal ExecuteEveryMinute
ExecuteEveryMinute={
    if timepattern(now(),"mm")="00"
        beep
        message "It's "+timepattern(now(),"hh")+ " o'clock"
    endif
}

```

Panorama will execute this procedure as close to the top of the hour as possible. If it cannot execute the procedure exactly at the top of the hour (because another program is on top, or you are editing data, graphics or a procedure) it will execute the procedure as soon as possible after the hour.

Triggering a Procedure As Soon As Possible

The [ExecuteASAP](#) variable is similar to the [ExecuteEverySecond](#) and [ExecuteEveryMinute](#) variables. To use this variable, create it and then fill it with the text of a procedure. The procedure will run as soon as possible when the current procedure is finished. This is especially handy in Handler procedures to allow them to trigger things they normally couldn't do, like open or close windows (see "[Event Handler Procedures](#)" on page 394).

The `ExecuteASAP` procedure should undefine the variable itself as part of the procedure. Otherwise the procedure will run over and over again. Just make sure the following line appears somewhere within the procedure.

```
undefine ExecuteASAP
```

To see an example of the `ExecuteASAP` variable in action, check out the `CLOSEWINDOWKEEPSECRET` statement in the `_UtilityLib` Library.

Event Handler Procedures

Panorama procedures are usually triggered by relatively "hi-level" events like clicking on a button or choosing from a menu. However there is a special type of procedure that is triggered by more low level events like simply bring a window to the front. These "event handler" procedures (also called simply "handler procedures") let you control how Panorama responds to these low level events.

Internally, event handler procedures work slightly differently than regular procedures. When an event handler procedure is triggered, Panorama stops everything and runs that procedure immediately. If a regular procedure causes the event handler procedure to trigger, the regular procedure will pause and wait for the event handler procedure to finish before continuing.

Event handler procedures are intended for changing the way Panorama responds to various low level events. An event handler procedure cannot contain any statements that would cause more low level events. In practical terms this means that an event handler procedure cannot change the arrangement of windows on the screen in any way—it cannot bring another window to the top, open a new window, close a window, or open or close any files. You cannot use the debugger with event handler procedures, because the debugger itself generates low level events. An error dialog will appear if your event handler procedure attempts to perform a statement that would cause another low level event. Event handler procedures should not pause for user input unless you really want to annoy your users. (However, it can sometimes be convenient to use a `message` statement to help debug an event handler procedure.)

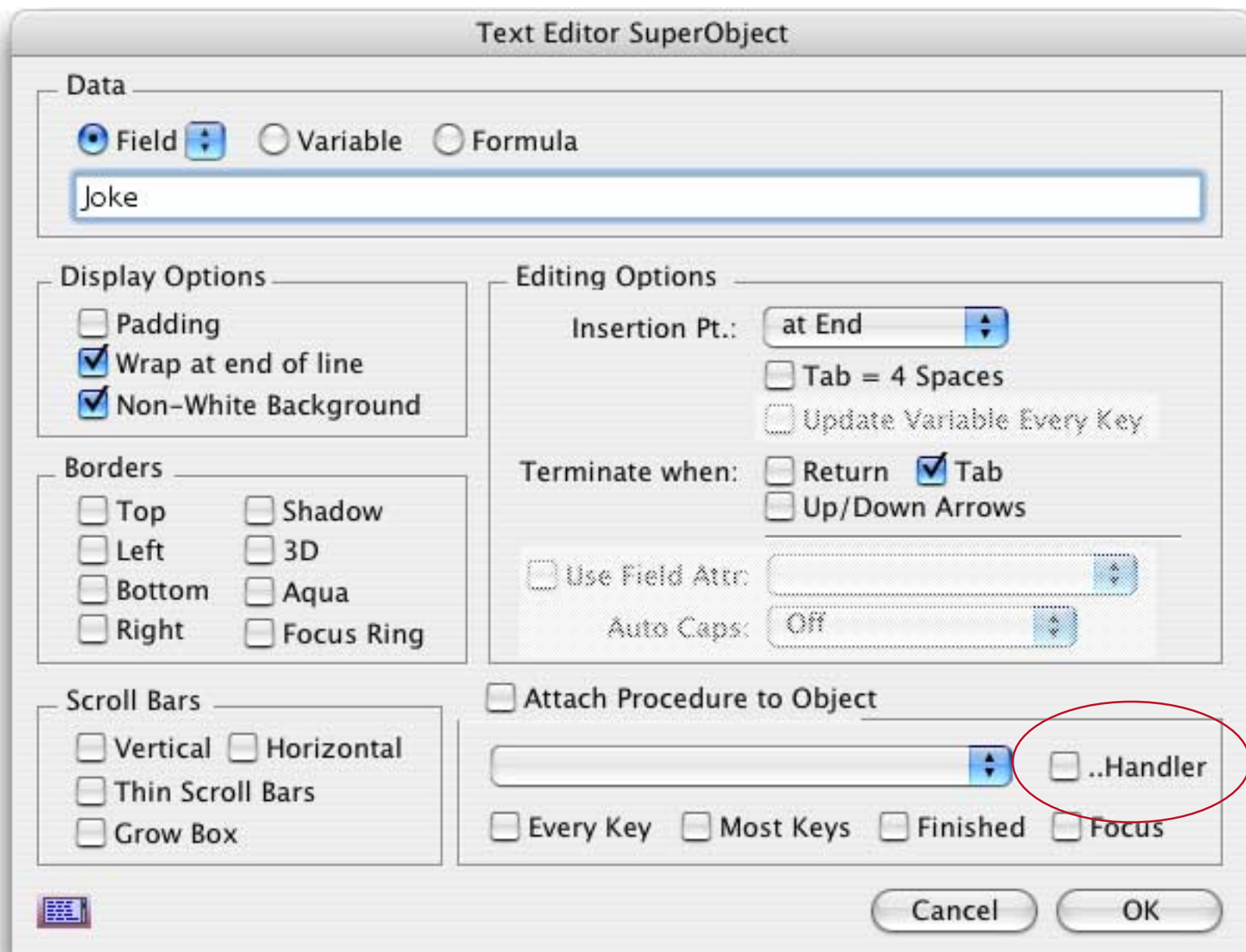
Event handler procedures should be as short as possible. Extra delays in processing low level events will be very noticeable. The event handler procedure should only deal with the event in question and should not contain any other logic for your application. If possible an event handler procedure should be written with one or two simple statements or formulas.

Sometimes the same procedure may be triggered in different ways — sometimes as a normal procedure, sometimes as an event handler procedure. In that case the procedure can use the `info("runninghandler")` function to determine what mode the procedure is currently running in. This function returns true if the procedure is running as an event handler, false if the procedure is running as a normal procedure.

Most event handler procedures have names that begin with two periods (`..ActivateForm`, `..CustomAbout`, etc.) to help distinguish them from ordinary procedures. The following sections describe each type of event handler procedure in detail.

Text Editor SuperObject ..Handler Option

The text editor SuperObject has always been able to trigger procedures when various events occur: pressing a key, pressing most keys, and terminating the editing of the object (see “[Text Editor Options](#)” on page 643 of the *Panorama Handbook*). However, users of early versions of Panorama encountered problems with this feature, since it would not work properly when other procedures were running, and would not work properly if the user terminated editing by clicking on another window. In addition, there was no way to automatically run a procedure when editing started. The **..Handler** option (in the Text Editor SuperObject Object Properties dialog) solves all of these problems.



When the **..Handler** option is turned on, all procedures triggered by the Text Editor SuperObject are treated as event handler procedures. The benefit of using event handler procedures is that these procedures are guaranteed to trigger and work properly under all conditions, no matter how the user started or stopped editing and whether or not another procedure is currently running. The only downside is that event handler procedures cannot open or close windows (see the previous section). To retain compatibility with databases created with earlier versions of Panorama you are allowed to turn the **..Handler** option off.

Focus Procedure

Panorama 3.1 added a new condition that may cause the Text Editor SuperObject to trigger a procedure. This condition is called **Focus**. When the **Focus** option is turned on, the text editor will trigger its procedure whenever you start editing that field. In other words, when you click on the field or tab into the field, the procedure will be triggered. When the procedure is triggered this way, the `info("trigger")` function will begin with **Focus**, followed by the name of the object, for example `Focus.TimeEditor` or `Focus.HTML`.

One use for a focus procedure is to implement Undo for editing. (Of course as of Panorama V this would be redundant, since Panorama already has Undo for editing within a cell.) Here is a procedure that saves the data in the field as the editing begins.

```
if info("trigger") beginswith "Focus."
  undoCell=«» /* «» is the current field */
  undoField=info("fieldname")
endif
```

The Undo procedure would look like this.

```
if undoField≠""
  set undoField,undoCell
endif
```

For completeness you may wish to add the following line to your **.CurrentRecord** procedure. This line ensures that you cannot undo after moving to a different record.

```
undoField=""
```

Another use for the **Focus** procedure is to memorize the selection point when editing was terminated and re-set the selection when editing resumes again. This example assumes that the database has two numeric fields named **textStart** and **textEnd**.

```
if info("trigger") contains "focus"
  activesuperobject "setselection",textStart,textEnd
else
  activesuperobject "getselection",textStart,textEnd
endif
```

The **Focus** option cannot be used if the **..Handler** option is turned off. This is not a big handicap, since you obviously don't want to change window just as editing begins. You should also keep the procedure as short as possible to minimize delay.

..OpenForm

The **..OpenForm** event handling procedure is triggered when any form in the same database as the procedure is opened. It doesn't matter how the form is opened—manually by the user, in a procedure, or automatically as part of opening the database. The most common uses for the **..OpenForm** procedure are initializing window variables and initializing SuperObjects. The **..OpenForm** procedure below will automatically open the Text Editor or Word Processing SuperObject named **Letter** when the **Editor** form is opened.

```
if info("formname")="Editor"
  superobject "Letter","open"
endif
```

..ActivateForm

The **..ActivateForm** event handling procedure is triggered when any form in the same database as the procedure is activated (brought to the front). It doesn't matter how the form is activated—manually (by clicking on it) or as part of a procedure (usually the **window** statement).

The example below shows how this procedure can be used with clone windows (see "[Window Clones](#)" on page 457). It assumes that your database contains a field called **ID** with unique values for each record. When a clone window is activated (brought to the front), the procedure automatically searches for the record that corresponds to that window.

```
windowvariable windowID
if info("formname") beginswith "Clone"
  find ID=windowID
endif
```

The careful reader may wonder if opening a form also activates it. The answer is yes. When a form is opened, both the **..OpenForm** and **..ActivateForm** procedures will be triggered (if they exist), in that order.

..CustomAbout

The **..CustomAbout** procedure allows you to change the name of the **About Panorama** item in the Apple menu (Mac) or Help menu (PC). This item normally says **About Panorama...** or **About Panorama Direct...**, but you can customize it to display any text you want when your database is active, for example **About This Database...**

The first step in customizing the **About** menu item is to create a new procedure in the database called **..CustomAbout**. You must spell this name exactly as shown, including upper and lower case.

The **..CustomAbout** procedure should only have a single statement in it: **SetAboutMenu**. This statement is followed by a text parameter, which could contain a formula, that specifies what text should be displayed. For example, if you want the Apple Menu to display **About Office 97...**, the **..CustomAbout** procedure should look like this:

```
setaboutmenu "About Office 97..."
```

Panorama runs the **..CustomAbout** procedure every time you click on a form or data sheet window. This means that you can adjust the Apple Menu for changing conditions. Here is a **..CustomAbout** procedure that displays the name of the current form:

```
setaboutmenu "About "+info("formname")+..."
```

The **..CustomAbout** procedure only applies to the forms and data sheet in the same database. As you click from window to window, the About item in the Apple Menu may change. When a procedure, flash art, or design sheet window is open the Apple Menu will display the standard **About Panorama...** or **About Panorama Direct...** message. The standard message will also be displayed when a window from any other database is active (unless that database also has a **..CustomAbout** procedure).

Note: By using the **..CustomAbout** and **.About** procedures (see **“.About”** on page 379) you can almost completely hide the fact that your application is created in Panorama. (However, on the Macintosh the Application Menu in the upper right hand corner of the screen will always show the name Panorama, while on the PC the master window and the task bar will always show the name Panorama.) Don't forget that you must include the Panorama copyright message in any custom About window that you create with the **.About** procedure!

..CloseDatabase

The **..CloseDatabase** procedure allows you to customize the actions that occur when a database is closed. Whenever a database is closed (either by clicking on the close box of the last open window, using the **Close File** command, or the **Quit** command), Panorama checks to see if the database contains a procedure called **..CloseDatabase**. If this procedure exists it will be called as a "handler" procedure.

If the procedure contains a **ContinueClose "yes"** or **ContinueClose "no"** statement, Panorama will not display the normal "Do you want to save changes" dialog, and will not save the database. Instead it is up to the procedure to decide what to do and to go ahead and do it. For example, if you just want to close the database without saving it the procedure can be simply one line:

```
ContinueClose "yes"
```

Please note that if the procedure contains a **ContinueClose "no"** statement, Panorama may ignore this under certain situations. For example, if a Quit has been requested by an Apple Event, the Quit will continue no matter what.

If necessary, the procedure can use the **info("quitinprogress")** to check whether Panorama is in the process of shutting down.

We've written some custom statements that automate the most common ways of handling database closing. The **CLOSEWINDOWKEEPSECRET** statement (in the `_UtilityLib` Library) doesn't allow Panorama to close the database. Instead, it makes the database secret (all windows closed) but leaves it in memory. However, the database will close when Panorama quits. This statement also saves the database. (Note: a `..CloseDatabase` procedure with this statement has been added to any custom statement library database created by the Custom Statement wizard. This is why the custom library is saved automatically when you close all of the windows.)

The **STANDARDCLOSEALERT** statement mimics the normal way Panorama closes a database. It displays the dialog asking the user whether they want to save, not save, or cancel, then does the appropriate operation. This statement is useful if you want the standard operation but with some additional processing before or after.

Chapter 3: Programming Techniques



The previous chapter covered the basics of working with procedures - how to create and edit procedures, set up and use variables, and basic control flow. This chapter builds on these basics and shows how to automate a wide variety of tasks within Panorama.

The structure of this chapter is a miniature version of the entire manual. We'll start with Chapter One and show how to automate file handling, then work our way through the entire manual showing how to automate all of the different operations available in Panorama.

Although this chapter is quite extensive, Panorama is so rich that there are many statements, functions and techniques that are not covered here. You'll find a gold mine of additional information in the Programming Reference wizard (see "[Programming Reference Wizard](#)" on page 237) and in the wizards and example files included with Panorama. You'll also find a very vibrant on-line community of Panorama developers on the "QNA" e-mail discussion list. See the tech support page on www.provue.com for information on subscribing to this list.

Accessing Files

A Panorama procedure has a wide variety of statements available for working with disk files. A procedure can open and close database files, import and export data to/from a database, or even read and write disk files directly (including resource files and registry entries).

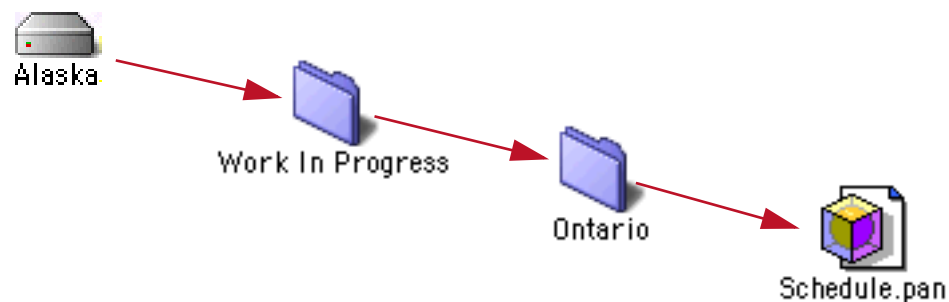
Files and Folders

On both Macintosh and Windows PC systems files are contained in folders, which may themselves be nested inside other folders. There are two different methods that Panorama uses to identify the name and location of a file. Method 1 is to combine both the name and location in a single string of text. Method 2 is to specify a separate file name and folder ID separately. Some statements and functions use method 1, some use method 2. Panorama also include functions that can convert back and forth between these two methods.

Combined Folder Location and File Name

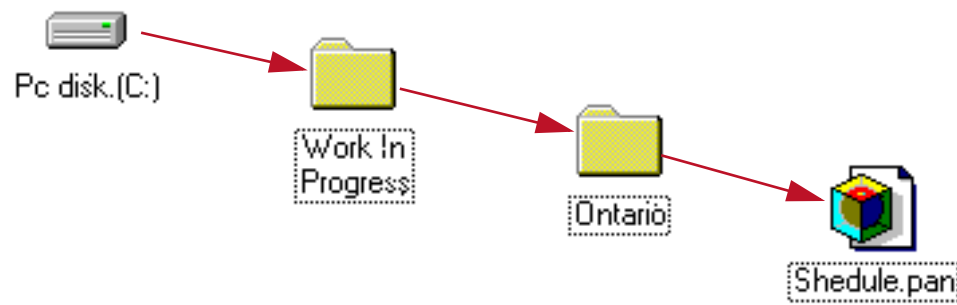
On the Macintosh, the exact location of any file can be specified by stringing together the name of the volume (disk) and the folders, each separated by a colon.

Alaska:Work in Progress:Ontario:Schedule.pan



Windows systems are similar, but backslashes (\) are used instead of colons, and drive names always consist of letters followed by a colon (A:, B:, C:, etc.).

C:\Work in Progress\Ontario\Schedule.pan

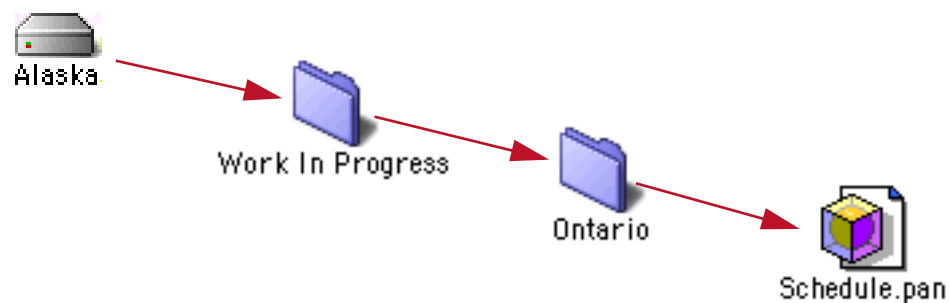


For cross platform compatibility, Panorama also allows you to use colons when using Panorama for Windows, like this:

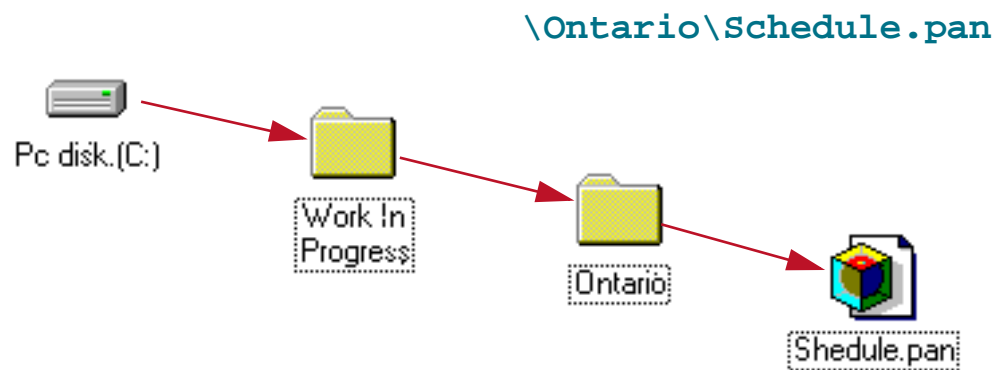
C::Work in Progress:Ontario:Schedule.pan

A file's location may also be specified relative to the current database. For example, suppose the current database was in the **Work in Progress** folder. In that case you could specify the location of the **Shedule.pan** file by simply leaving off left hand portion of the specification. The specification must begin with a colon or backslash to indicate that it is relative to the current folder and not an absolute location.

:Ontario:Schedule.pan



On PC systems you can specify this relative location like this:



However, keep in mind that on PC systems Panorama will accept `:` instead of `\`. Therefore, the specification

```
:Ontario:Schedule.pan
```

will work on both Windows and MacOS based computers. You should use colons if your database might ever be used on both Macintosh and Windows computers.

Folder ID's and Paths

A folder's **path** is simply a list of folders within folders, each separated by a colon (Macintosh) or backslash (Windows PC). The list always starts with the name of the disk drive. Here are some examples of folder paths:

```
GigDrive:Work In Progress:
```

```
F:\Clients\Taco Bell\
```

```
HD:System Folder:Preferences:
```

```
C:\Windows\
```

A **folder ID** is a 6 byte binary value that uniquely identifies a folder. Folder ID's are used by several of the Panorama functions and statements. Panorama can convert a path to a folder id with the `folder()` function, like this:

```
folder("MyDisk:Clients:")
```

The `folderpath()` function converts a folder ID back into a path, for example:

```
folderpath(dbinfom("folder"), "Checkbook")
```

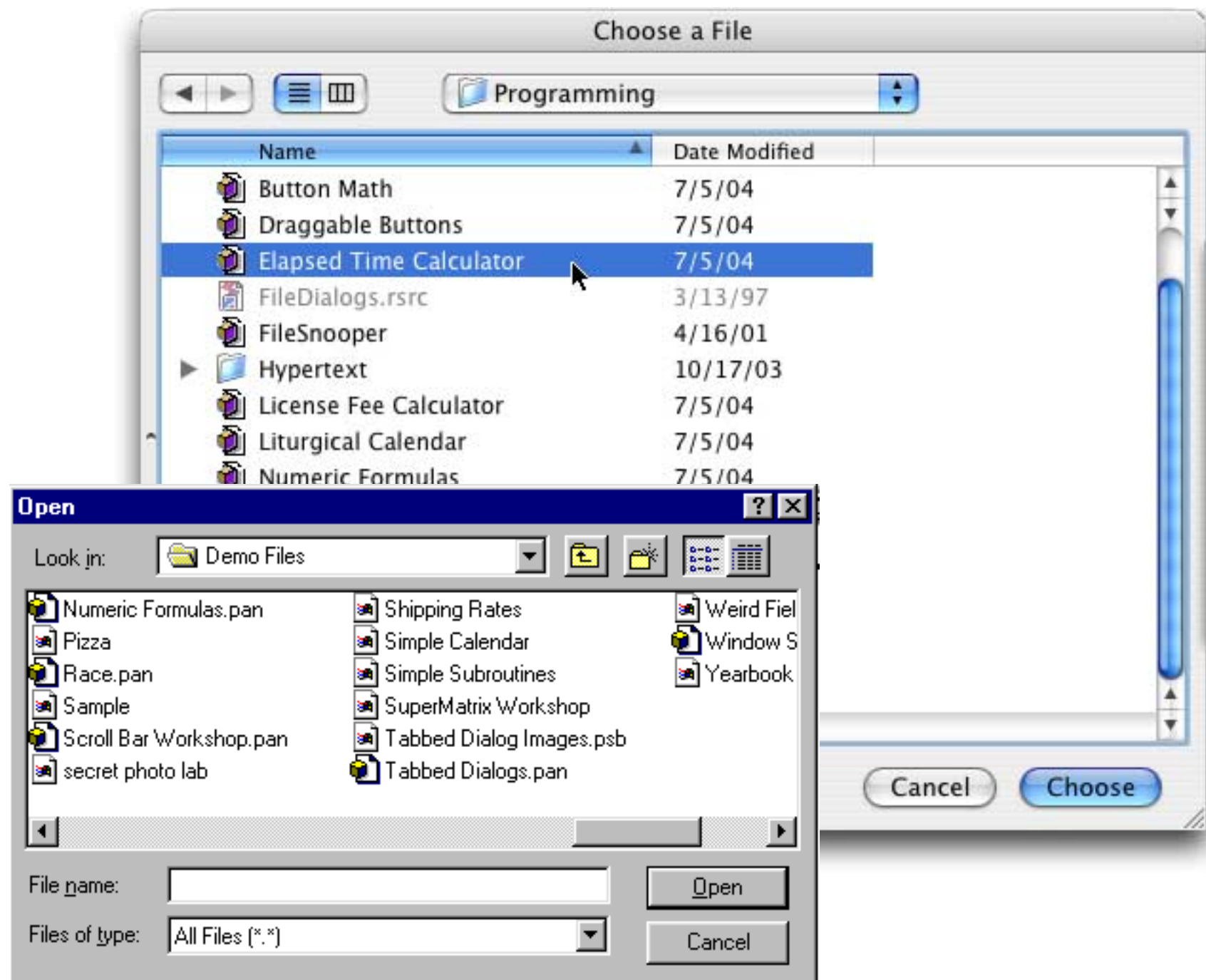
Panorama has numerous functions for working with folders and paths. See "[Disk Files and Folders](#)" on page 165 for detailed information on these functions.

Tip: Most functions or statements that use a folder ID will accept `"`, which means "the folder that contains the current database."

Locating a File with Standard Dialogs

Both the Macintosh and Windows PC's have standard dialogs for locating a file. With the `openfiledialog` and `savefiledialog` statements, your procedures can use these dialogs also.

The `openfiledialog` statement allows the user to select the file using a standard file open dialog (see "[OPENFILEDIALOG](#)" on page 5574 of the *Panorama Reference*). This illustration shows what this dialog looks like on both Windows PC and Macintosh computers (the exact appearance may vary depending on what operating system version and extensions you are using).



The file may be anywhere on any disk drive mounted on the computer. The `openfiledialog` statement has four parameters.

```
openfiledialog folder,file,type,typelist
```

The `folder` parameter is a folder ID (see "[Folder ID's and Paths](#)" on page 401). This specifies what folder should be listed when the dialog first opens.

The `file` parameter is the name of the file the user selected. If this parameter is empty the user pressed the **Cancel** button.

The `type` parameter is the type of the file the user selected. This is a four letter descriptor, for example TEXT for text files or ZEPD for Panorama database files.

The `typelist` parameter specifies what kind of files you want listed in the file dialog. If this parameter is empty (`""`), then all files will be listed. Otherwise, this should be one or more 4 letter “file type” descriptions. Here are two file type descriptions you may find useful.

Type	Description
ZEPD	Panorama database
TEXT	Text file

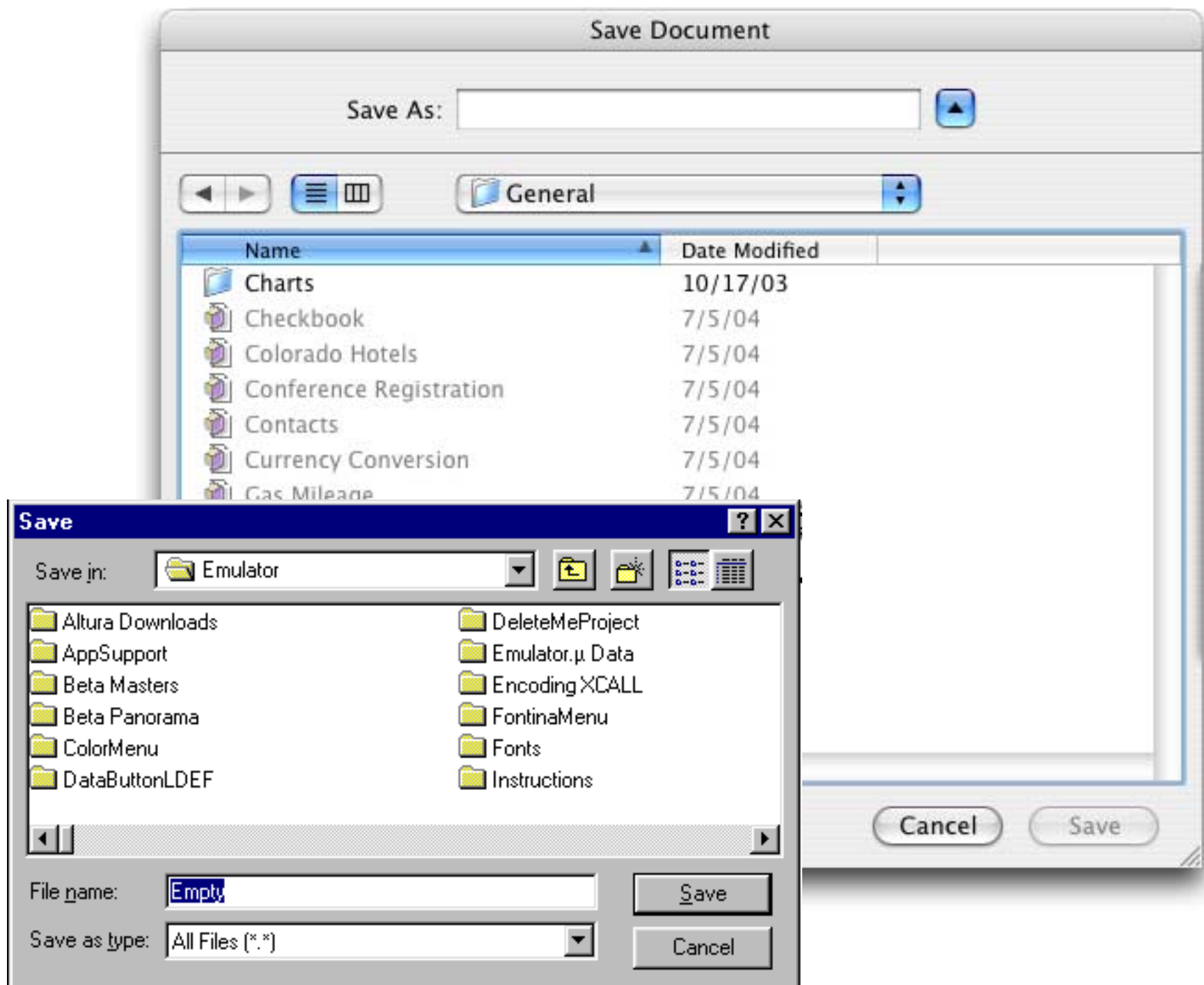
The file type must be entered exactly as shown. Only files whose types match exactly (including upper/lower case) will be shown.

Here is an example of a procedure that allows the user to select a Panorama file, then appends that file to the current file.

```
local fileFolder,fileName,fileType
openfiledialog fileFolder,fileName,fileType,"ZEPD"
if fileName="" stop endif ; user pressed cancel
openfile "+"+folderpath(fileFolder)+fileName
```

To change this example so that it can be used for importing text files, change `"ZEPD"` to `"TEXT"`. Or, to allow either choice, change this parameter to `"ZEPDTEXT"` (or `"TEXTZEPD"`).

The `savefiledialog` statement allows the user to select a file using a standard file save dialog (see “[SAVE-FILEDIALOG](#)” on page 5698 of the *Panorama Reference*). This illustration shows what this dialog looks like on both Windows PC and Macintosh computers (the exact appearance may vary depending on what operating system version and extensions you are using).



The user may type in the name of the file, and can select what folder the file will be saved into. The file may be anywhere on any disk drive mounted on the computer. The `savefiledialog` statement has three parameters.

```
savefiledialog folder,file,prompt
```

The `folder` parameter is the ID of the folder the user selected.

The `file` parameter is the name of the file the user entered. If this parameter is empty, the user pressed the `~` button.

The `prompt` parameter is a short phrase that you supply. When using MacOS Panorama will display this phrase in the dialog title, for example [Enter file name](#).

Here is an example of a procedure that allows the user to specify a file name and location, then exports to that file.

```
local fileFolder,fileName
fileName=""
savefiledialog fileFolder,fileName,"Export file name:"
if fileName="" stop endif ; user pressed cancel
export folderpath(fileFolder)+fileName,replace(exportline(),␣,chr(11))+␣
```

If the user specifies a file that already exists, Panorama will warn them and ask if they really want to erase the existing file.

Customizing the Standard File Dialogs

Versions of Panorama prior to 5.0 allowed you to customize the dialog that appears when the `openfiledialog` and `savefiledialog` statements are used. However, this only worked on Macintosh computers, and only with the older OS 6/7 style dialogs. Since Panorama now uses Apple (relatively) new File Navigation Services system, the open and save dialogs can no longer be customized on either the Macintosh or on Windows.

Opening a Panorama Database

To open a Panorama database use the `openfile` statement (see “[OPENFILE](#)” on page 5572 of the *Panorama Reference*). If the database to be opened is in the same folder as the current database you can simply supply the filename.

```
openfile "Contacts"
```

If the file is in a different folder you must supply a combined path and file name (see “[Combined Folder Location and File Name](#)” on page 400).

```
openfile "C:\My Documents\Organizer\Contacts"
```

On a Windows PC system the file would actually be named `Contacts.pan`, but you may omit the `.pan` from the name.

Here is an example of a procedure that allows the user to select a Panorama file and then open it (see “[Locating a File with Standard Dialogs](#)” on page 402).

```
local fileFolder,fileName
openfiledialog fileFolder,fileName,"ZEPD"
if fileName="" /* user pressed cancel */
  stop
endif
openfile folderpath(fileFolder)+fileName
```

Suppressing the Default Extension

On Macintosh systems the behavior of the `openfile` statement changes slightly if the current database name ends with `.pan`. In that case the `openfile` statement will automatically add `.pan` to any file name that doesn't already have an extension. This makes it easier to set up a set of database files that can work on both the Macintosh and the PC. However, this also means that you will not be able to open a database that doesn't have any extension at all with this statement (of course you can always open such a file manually using the `Open File` dialog in the File menu).

If you need to use the `openfile` statement to open a file without an extension when the current file does have the `.pan` extension you must use the `nodefaultextension` statement just before the `openfile` statement. Here is a modified version of the procedure from the last section that allows the user to select any Panorama file (with or without the extension) and then open it.

```
local fileFolder,fileName
openfiledialog fileFolder,fileName,"ZEPD"
if fileName="" /* user pressed cancel */
    stop
endif
nodefaultextension
openfile folderpath(fileFolder)+fileName
```

Appending Databases End-to-End

If you have two database files with identical field structures (the same fields in the same order), it's easy to append them together. Essentially appending sticks the second database right onto the end of the current database. To append a database use the `openfile` statement, but put a `+` symbol in front of the name of the database. The example below appends the file `MoreCustomers` to the end of the current database. (Note: The `MoreCustomers` file does not have to be open, but it's ok if it is.)

```
openfile "+MoreCustomers"
```

If the file you want to append doesn't have identical field order, but the field names are the same, use two `+` symbols, like this:

```
openfile "++MoreCustomers"
```

Only fields with matching names will be appended together (this is the same as checking the **Match Fields by Name** option in the **Open File** dialog, see "[Appending One Database to Another](#)" on page 78 of the *Panorama Handbook*). If there are any extra fields in the current database that do not match, they will be left blank in the new appended data.

The `openfile` statement can also be used to append a text file to the current database. See "[Importing Text Files](#)" on page 409.

Eliminating Duplicates in Appended Data

If you need to eliminate duplicates between the newly appended data and the original data, you can use the `sortup` statement to bring the duplicates together, followed by the `unpropagate` statement to clear out the duplicate company names (see "[Using UnPropagate to Eliminate Duplicates](#)" on page 470 of the *Panorama Handbook*). The `select` and `removeunselected` statements actually eliminate the duplicate records. The example below appends to the customer file, then eliminates any duplicates that were appended. Newly appended records that are not duplicates of the existing data will be kept.

```
openfile "+MoreCustomers"
field Company
sortup
unpropagate
select Company<>" "
removeunselected
```

If you want to keep the newly appended data and remove the original data when a duplicate is found, use the `unpropagateup` statement instead of the `unpropagate` statement. For example, you would do this if you felt the newly appended data was more up-to-date or more reliable than the original data.

Replacing the Data in a Database

Occasionally you may want to throw out all the data in the current database and replace it with the data in another database. For example, the data in the current database may be out-of-date. To do this use the `openfile` statement and put an `&` symbol in front of the file name:

```
openfile "&NewCustomerList"
```

You may be wondering, if the data is already in a database, “Why not use that database instead of replacing the data in the current database?” Usually it’s because the file that contains the correct data does not have the forms and procedures set up they way you want them. You can easily transfer the data, transferring forms and procedures is a much more tedious job.

If the database file you want to load doesn’t have the fields in the same order, but the field names are the same, use two `&` symbols in a row.

```
openfile "&&NewCustomerList"
```

Only fields with matching names will be loaded with data (this is the same as checking the **Match Fields by Name** option in the **Open File** dialog, see “[Appending One Database to Another](#)” on page 78 of the *Panorama Handbook*). If there are any extra fields in the current database that do not match, they will be left blank in the newly loaded data.

The `openfile` statement can also be used to replace the current data with data from a text file (see “[Importing Text Files](#)” on page 409). The `&` option becomes very handy because you can load the raw text into a pre-prepared database that contains ready-to-use forms and procedures.

Saving a Panorama Database

To save the current database use the `save` statement (see “[SAVE](#)” on page 5692 of the *Panorama Reference*). This procedure adds a new record and then saves the database.

```
addrecord
save
```

To save all open databases use the `saveall` statement (see “[SAVEALL](#)” on page 5694 of the *Panorama Reference*).

To save the current database with a new name or in a new location use the `saveas` statement (see “[SAVEAS](#)” on page 5695 of the *Panorama Reference*). The new file becomes the current version of the file. The procedure below will save the current file under the name **Monday**, **Tuesday**, **Wednesday**, etc. depending on the day of the week.

```
saveas datepattern(today(), "DayOfWeek")
```

The `saveacopyas` statement also saves the database with a new name or location, but the new file does not become the current version of the file (see “[SAVEACOPYAS](#)” on page 5693 of the *Panorama Reference*). This procedure saves a backup copy of the current file in a folder named **Backups**.

```
saveas "C:\Backups\"+info("databasename")
```

A procedure can also save the current file as a text file, this is called **exporting** the file. See “[Exporting Text Files](#)” on page 413 to learn how to do this in a procedure.

A procedure can use the `revert` statement to revert to the last previously saved copy of the database (see “[REVERT](#)” on page 5676 of the *Panorama Reference*).

Closing a Database

To close the current database use the `closefile` statement (see “[CLOSEFILE](#)” on page 5109 of the *Panorama Reference*). All windows associated with the current file will also be closed.

If the `closefile` statement is in the middle of a procedure it will immediately close the file without stopping to ask if you want to save the file. For example, this procedure will close the `Checkbook` and `Payments` databases immediately, without saving them, and then open the `Bills` database.

```
window "Checkbook"  
closefile  
window "Payments"  
closefile  
openfile "Bills"
```

A safer version of this procedure might look like this.

```
window "Checkbook"  
save  
closefile  
window "Payments"  
save  
closefile  
openfile "Bills"
```

There is one exception to this rule. If the `closefile` statement is the very last statement in the procedure Panorama will stop and ask if the file should be saved. If you don't want that to happen, add a `nop` statement (see "[Doing Nothing for a While](#)" on page 280) after the `closefile` statement, like this.

```
closefile  
nop
```

Shutting Down Panorama

To shut down Panorama use the `quit` statement. The `quit` statement will not normally ask the user if they want to save changes in any open databases before stopping Panorama. However, if the `quit` statement is the last statement in the procedure, or is followed by a `stop` statement, it will ask the user if he or she wants to save each file, and if they say **yes**, save the files for them.

Importing Text Files

Panorama cannot directly access information in files created by other database or spreadsheet programs. Exchanging information between Panorama and other programs requires an intermediate file which is called a “text” or “ASCII” file (see “[Working with Text Files](#)” on page 81 of the *Panorama Handbook*). A text file is very basic because it contains just the data—no forms, procedures, formulas, pictures or anything else. Because text files are so simple they provide a common interchange format for different programs. Virtually all database, spreadsheet, and word processing programs on both the Macintosh and the PC can read and write text files.

Like the Panorama data sheet, a text file is divided into records and fields. Each line in the text file is a single record. The fields may be separated by commas or by tabs. You will often hear that a file is “tab delimited” or “comma delimited,” because the tab or comma character delimits the boundaries between each field. When you export from Panorama you must specify whether to use commas or tabs. When Panorama imports a text file it automatically checks for tabs. If it doesn’t find any, it assumes that this text file is comma delimited.

Carriage Returns in the Data

One complication with using text files is handling carriage returns that are actually part of the data in a cell. Usually a carriage return signals the end of the record and the start of a new record. If a carriage return is in a cell, Panorama thinks a new record is starting, and the rest of the import will be misaligned.

The best solution to this problem is to convert carriage returns in cells into vertical tabs (ASCII value 11) in the exported text file. Many programs do this automatically. When the data is imported into Panorama the vertical tabs are automatically converted back into carriage returns again.

If you are manually exporting data from Panorama (using the **Save As** command, see “[Exporting a Text File](#)” on page 105 of the *Panorama Handbook*) and want the carriage returns converted to vertical tabs, make sure the **Tabs w/o quotes option** is turned on, and the **Output Patterns** option is turned off.

Importing a Text File into an Existing Database

The `openfile` statement can also be used to import a text file into the current database (the text can either be appended to or replace the current data). This is called **importing** the text. In a procedure, the technique for importing text is exactly the same as appending or replacing two Panorama databases. Panorama checks the file you have specified to see if it is a database or text file, and automatically imports it if it is a text file.

This example will append the data in a text file named `RawData.txt` to the end of the current database.

```
openfile "+RawData.txt"
```

This example will erase all the data in the current database, then import the data in a text file named `NewRawData.txt`.

```
openfile "&NewRawData.txt"
```

Both of these examples have shown files with the `.txt` extension. On the Macintosh this is optional. On Windows PC systems this extension is required. If you need to import a text file that does not have the `.txt` extension you can use the `opentext` statement (see “[OPENTEXTFILE](#)” on page 5585 of the *Panorama Reference*). This statement is the same as the `openfile` statement except that it treats all files as text files, no matter what extension they have.

The text file must be separated into columns with either tabs or commas, and into rows with carriage returns or carriage return/line feed pairs. (If the first line of the file contains a tab Panorama assumes the file is tab delimited, otherwise comma delimited.) If any vertical tab characters are encountered (character value: 11) they are converted to carriage returns within the actual cell (in other words, this allows you to import carriage returns within a cell without starting a new record.)

Importing from a Variable

If the filename begins with the @ symbol the `openfile` statement will import from a variable instead of from a text file. For example, the procedure below will import the text contents of the variable `MyData` into a new database.

```
openfile "@MyData"
```

Adding the + symbol causes the text contained in the variable to be appended to the end of the current database.

```
openfile "+@MyData"
```

Adding the & symbol causes the text contained in the variable to replace all of the data in the current database.

```
openfile "&@MyData"
```

A useful technique is to build an array of data from one database (see “[Building an Array from a Database](#)” on page 594) and then import that data directly into another database.

Importing HTML Tables

When the `openfile` statement imports a text file (or variable) it checks the text to see if it contains one or more HTML `<table>` tags. If a `<table>` tag is found the other text is ignored and Panorama simply imports the data inside the table. The text will be divided into rows and columns based on the `<tr>` and `<td>` tags within the table. Here’s an example of how to import a table.

```
openfile "Financial_News.html"
```

If the text file (or variable) contains more than one table only the first will be imported. Any additional tables will be ignored. If you need to import a different table the procedure can use the `fileload()` function to read the text into a variable (see “[Reading Data Files](#)” on page 422), then use the `tagdata()` function to extract the table you want to import (see “[Tag Parameter Functions](#)” on page 103). The example procedure below imports the 2nd table from the HTML file (instead of the 1st).

```
local rawHTML,theTable
rawHTML=fileload("","Financial_News.html")
theTable="<table"&tagdata(rawHTML,"<table","</table>",&2)+"</table>"
openfile "@theTable"
```

Re-Arranging the Order of Imported Data

Panorama normally imports text directly into the database, column for column. In other words, the first column in the text file goes into the first field in the database, the second column into the second field, etc. This is great if the text file is set up the same way as your database. If not, you’ll need to process the data as it comes in, possibly re-arranging and combining data as you go. This processing is done with a formula you design. The formula is set up by the `importusing` statement (see “[IMPORTUSING](#)” on page 5355 of the *Panorama Reference*), which should be the statement immediately before the `openfile` statement. The `importusing` statement has one parameter, the formula for processing the data. Here’s the general idea of how these statements must be used together (several more specific examples follow below).

```
importusing formula
openfile "+MyTextFile.txt"
```

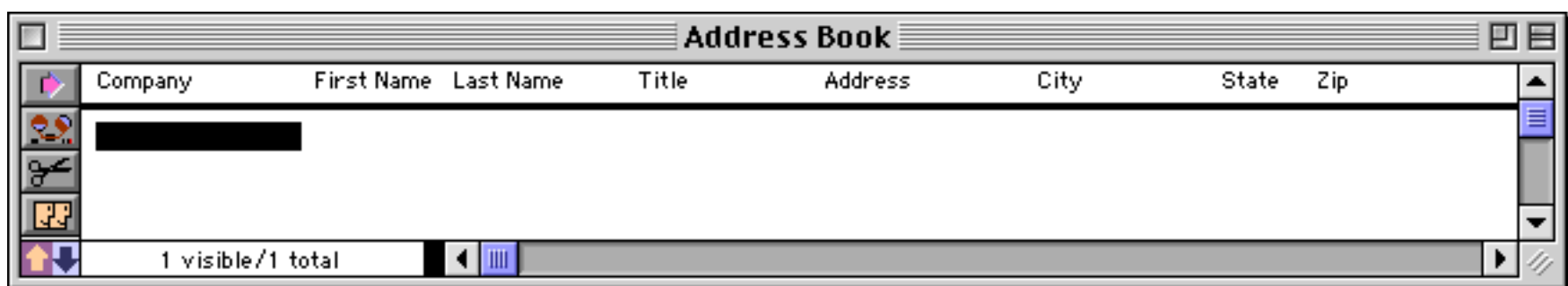
By itself, the `importusing` statement doesn't do anything except stash the formula you provide where the `openfile` statement can find it. Here's what happens. First `openfile` statement reads one line from the text file. But it doesn't actually import the line into the database. Instead, it evaluates the `importusing` formula. The formula takes the line of data and re-arranges it. The `openfile` statement then takes the result of this formula and imports that result into the database. This process is repeated over and over again for each line in the text file: read, calculate, import.

To process the line that the `openfile` statement has read in, the formula needs to be able to access the data in that line. There are two special functions that allow you to read this line. The `import()` function (see "[IMPORT\(\)](#)" on page 5352 of the *Panorama Reference*) accesses the entire line that has been read in. The `importcell(columnNumber)` function (see "[IMPORTCELL\(\)](#)" on page 5353 of the *Panorama Reference*) accesses an individual cell in the line (tab or comma delimited). The `columnNumber` starts with 0 for the first column, 1 for the second column, etc.

Suppose you have a text file named `Sam's Contacts.txt` that contains data like this (each column of data represents fields separated by a tab):

```
Smith   John   World Widgets      124 W. Olive St   San Jose  CA  95134
Lee     Susan  Industrial Metals   2347 N. Riverside Cambridge MA 02139
Marklee Lance  Zipper Technologies 687 E. Dorothy Lane Bothell WA 98011
Anders  Fred   Acme Fireworks     5672 Lakewood Drive Salinas CA 93908
```

You want to import this data into a database that contains these fields:

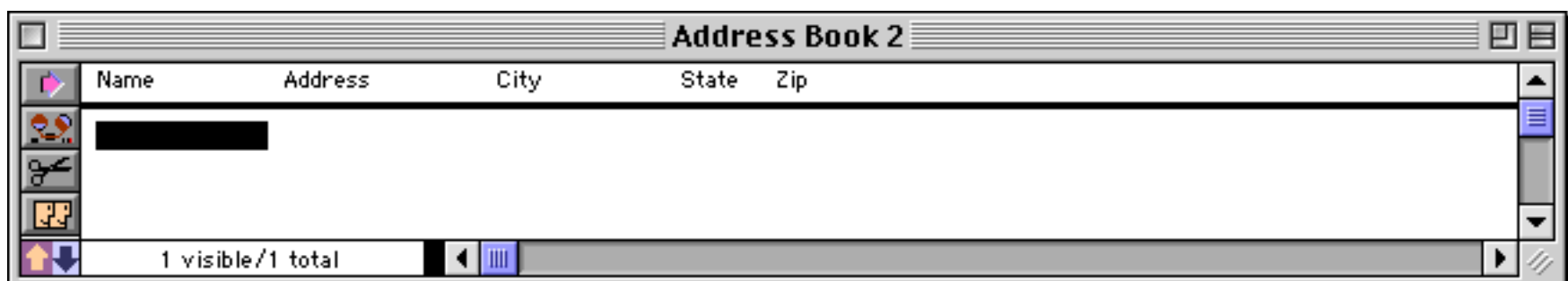


Here's a procedure that will append the data in `Sam's Contacts.txt` into this database. The tabs (–) in the formula divide the output into separate columns again so it can be imported (see "[Special Characters](#)" on page 57).

```
importusing importcell(2)+--+importcell(1)+--+importcell(0)+--+--+
importcell(3)+--+importcell(4)+--+importcell(5)+--+importcell(6)
openfile "+Sam's Contacts.txt"
```

The formula re-arranges the incoming data so that third column in the input text goes into the first field, the 2nd column goes into the 2nd field, the first column goes into the 3rd field, the 4th field is empty, the 4th column goes into the 5th field, the 5th column goes into the 6th field, the 6th goes into the 7th field and the 7th column goes into the 8th field.

In this example, each column in the input corresponds with one field in the final database. However, you could split up a column into multiple fields, or combine multiple columns in the input text into a single field in the final database. For example, suppose your `Address Book` database only had five fields, like this.



Here is a procedure that imports **Sam's Contacts.txt** into this version of the **Address Book** database.

```
importusing importcell(1)+" "+importcell(0)+--+
importcell(3)+--+importcell(4)+--+importcell(5)+--+importcell(6)
openfile "+Sam's Contacts.txt"
```

This formula simply concatenates the first and last names with a space, but you can use any function you want, including the **?**(, **sandwich**(, **upper**(, **lower**(, even **lookup**(functions.

Building the ImportUsing Formula on the Fly

The **importusing** formula is usually hard-coded into the procedure. What if however, you don't know how the data should be re-arranged in advance? In that case you might want to design a dialog that would allow the user to configure how the data is re-arranged. The dialog would build the formula for importusing. To pass the formula to the importusing statement, you must put the formula into the clipboard and then use the following statement:

```
importusing clipboard
```

Warning: You must enter this statement exactly as shown here. Do not put **()** after the word **clipboard**. Do not add anything else at all to this statement.

Here is a very simplified example that shows how it is done. First the procedure asks for the name of the file to import, then it asks for the order of the fields. The field order should be entered as numbers separated by spaces, for example **4 1 2 3 8 9 10**.

```
local importFile,importOrder,importFormula,X
importFile="" importOrder="" importFormula=""
gettext "Import what file?",importFile
gettext "Field order? (ex: 4 1 2 3)",importOrder
X=1
loop
importFormula=sandwich(" ",importFormula,"+--+")+
importcell("+array(importOrder,X," ")+"")
X=X+1
while X<=arraysize(importOrder," ")
clipboard=importFormula
importusing clipboard
openfile "+"+importFile
```

As the program goes through the loop it builds up the formula. For example, if the user entered **3 2 4** the procedure would generate the formula:

```
importcell(3)+--+importcell(2)+--+importcell(4)
```

When the formula is completely assembled it is placed into the clipboard, and then the text file is appended to the current file.

Although this example illustrates how the technique works, the user interface is lousy. Instead of having the user type in the field order, you'll probably want to let the user choose the list order with a List or Pop-Up SuperObject.

Exporting Text Files

You can manually export a database to a text file with the **Save As** command (see “[Exporting a Text File](#)” on page 105 of the *Panorama Handbook*). To export a database from a procedure, use the **export** statement. This statement has two parameters:

```
export file,formula
```

The **file** parameter is the name of the text file to export. If this file already exists, it will be erased and the new file will replace it. If the file needs to be in a different folder than the current database then a combined folder and file must be supplied (see “[Combined Folder Location and File Name](#)” on page 400).

The **formula** parameter is a formula that controls how the data is formatted as it is exported. To do its job, the **export** statement scans each visible (selected) record in the database. For each record it calculates the result of the formula, and adds that result to the text file being exported.

Usually the export formula consists of a series of fields separated by tabs and ending with a carriage return. Remember, in a formula `↵` represents a tab and `␣` represents a carriage return (see “[Special Characters](#)” on page 57).

Here is a typical formula that exports a name and address list.

```
export "Addresses.txt",Name↵↵Address↵↵City↵↵State↵↵Zip↵␣
```

Numeric and date fields must be converted to text before they can be exported. The functions listed in this table can perform these conversions.

Function	Reference Page	Description
exportcell(value)	Page 5209	This function converts a value into text without any special formatting. For numeric values this function is the same as the str (function (see below). The advantage of this function is that it works with any kind of value - text, numeric or date. Use this function when for some reason you don't know what kind of data you need to convert.
pattern(number,string)	Page 5599	This function converts a number into text, using the string as an output pattern. For example the formula pattern (Price,"\$#.,##") will convert the price 3458.23 into the string \$3,458.23. The pattern adds the \$ and the comma. For more information on numeric output patterns see “ Numeric Output Patterns ” on page 250 of the <i>Panorama Handbook</i> .
str(number)	Page 5799	This function converts a number into text without any special formatting. If you want to format the number (add commas, set # of digits, etc.) use the pattern (function.
datepattern(number,pattern)	Page 5148	This function converts a number representing a date into a formatted text string. The pattern parameter is an output pattern telling the function how to format the date. For more information on date output patterns, see “ Date Output Patterns ” on page 255 of the <i>Panorama Handbook</i> . Use the datepattern (function to store a date in a text field, or to display a formatted date in an auto-wrap text object or Text Display SuperObject. For example, the formula: <pre>datepattern(«Ship Date», "Month ddnth, yyyy")</pre> can be used to display the date an order was shipped in the format May 12th, 2003 .

Here is a function that exports data from a checkbook database using these functions.

```
export "Checks Archive.txt",str(«Check#»)+↵↵datepattern(CheckDate, "MM/DD/YY")+↵↵PayTo+↵↵Category+↵↵str(Debit)+↵↵str(Credit)+↵␣
```

If one or more data cells might contain carriage returns, you may wish to convert the carriage returns into vertical tabs as they are being exported. The generic formula for this conversion is `replace(<text>,"",chr(11))` (see “[REPLACE\(\)](#)” on page 5665 and “[CHR\(\)](#)” on page 5099 of the *Panorama Reference*). Here’s a specific example.

```
export "Addresses.txt",replace(Name--+Address--+City--+State--+Zip,"",chr(11))+
```

If you simply want to export all the fields in the same order that they appear in the data sheet, use the `exportline()` function (see “[EXPORTLINE\(\)](#)” on page 5210 of the *Panorama Reference*). This function produces tab delimited output of all the fields.

```
export "GenericText",replace(exportline(),"",chr(11))+
```

Don’t forget the `␣` on the end of the formula. Without this the exported text file will all be on a single line!

Exporting Line Items as Separate Records

Usually when Panorama exports data there is one line in the exported text file for each selected record in the database. So if the database has 67 records, the exported text file will have 67 lines.

This one-to-one correspondence does not apply if the export formula contains one or more line item fields with the Ω symbol (see “[Special Characters](#)” on page 57), for example `Qty Ω` or `Price Ω` . If the formula contains line items, the export file will contain multiple lines for each record—one line for each line item.

For example, consider an invoice database with 6 line items: `Qty1`, `Qty2`, ... `Qty6`, `Description1`, `Description2`, ... `Description6`, `Price1`, `Price2`, ... `Price6` etc. The procedure below can be used to export the line items from this database:

```
export "Items",<Invoice#>--+
  str(Qty $\Omega$ )--+Description $\Omega$ --+str(Price $\Omega$ )--+str(Total $\Omega$ )+
```

The text file (`Items`) will contain 6 lines for each invoice. The first line will contain the invoice number, `Qty1`, `Description1`, `Price1`, and `Total1`. The second line will contain the invoice number, `Qty2`, `Description2`, `Price2`, and `Total2`. The output continues for each line item, then starts over at `Qty1` for the next record.

Here’s a sample of how the exported data might look. This shows the data from three invoice records. `Invoice 17882` has three line items. The remaining 3 line items are blank. The next two invoices have two line items each.

```
178822  Widget          4.00  8.00
178821  Mini Widget      2.50  2.50
178824  Modern Widget    5.00  20.00
17882
17882
17882
178831  Art Deco Widget  7.50  7.50
178833  Thingy           3.00  9.00
17883
17883
17883
17883
178841  Modern Widget    5.00  5.00
178842  Micro Thingy     4.00  8.00
17884
17884
...
...
```

Warning: The line item export feature will only work if all the line item fields in the database have the same number of line items. If some line item fields have more line items than others, only a single record will be exported. For example if the database contains [Address1](#), [Address2](#), [Qty1](#), [Qty2](#), [Qty3](#), [Qty4](#), [Qty5](#) the line item export feature will not work. The solution is to rename [Address1](#) and [Address2](#) to names that don't end with a number (perhaps [Address](#) and [Suite](#), for example).

Analyzing Line Items

One great application for exporting line items as separate records is that you can re-import them into another database and analyze them. For example, from the invoice database it is very difficult to find out how many widgets or doo-dads you've sold. The information is split across all the line item fields. But if you export the line items as separate records and re-import this file into another database, it's easy to sort or select line item data.

The procedure below assumes you have set up a database called Line Item Analysis in advance. This database has five fields: [Invoice#](#), [Qty](#), [Description](#), [Price](#) and [Total](#). The procedure exports the line item data from the [Invoice](#) database, then re-imports it into the [Line Item Analysis](#) database.

```
export "Items",«Invoice#»+--+str(QtyΩ)+--+DescriptionΩ+--+str(PriceΩ)+--+str(TotalΩ)+¶
window "Line Item Analysis"
openfile "&Items"
select Description≠" "
removeunselected
field Description groupup
field Qty total
field Price average
field Total total
outlinelevel 1
```

After the data is imported, the procedure removes all of the empty records (with the [select](#) and [removeunselected](#) statements). Then the procedure uses Panorama's standard analysis tools ([groupup](#), [total](#), [average](#), etc.) to calculate how many of each type of item has been sold at what average price.

Exporting Array Elements as Separate Records

Usually when Panorama exports data there is one line in the exported text file for each selected record in the database. So if the database has 67 records, the exported text file will have 67 lines.

This one-to-one correspondence does not apply if the export formula contains one or more [arrayscan\(\)](#) functions. This function allows you to export the contents of arrays (see "[Text Arrays](#)" on page 93) with one element per exported line. The [arrayscan\(\)](#) function (see "[ARRAYSCAN\(\)](#)" on page 5052 of the *Panorama Reference*) has two parameters:

```
arrayscan(field,separator)
```

The [field](#) parameter is the name of a database field that contains an array. (A variable will also work, but usually doesn't make sense.) The [separator](#) parameter is the separator character for this array (see "[Picking a Separator Character](#)" on page 93).

For example, suppose your database has a [Phones](#) field which contains an array of one or more phone numbers, separated by a carriage return. Each array element contains the type of phone number, a comma, and the phone number itself, like this:

```
home,(714) 555-1212
office,(714) 555-8932
fax,(714) 555-8938
```

The procedure below will export the phone numbers with one record per phone number:

```
export "Phone List",
Name+--+array(arrayscan(Phones,¶),1,"")+--+array(arrayscan(Phones,¶),2,"")+¶
```

This procedure will output a text file something like this:

```
Joan Selbyhome(714) 555-1212
Joan Selbyoffice(714) 555-8932
Joan Selbyfax(714) 555-8938
Sally Rogersoffice(508) 777-8922
Sally Rogersfax(508) 777-8910
Chris Robertsoffice(909) 874-1234
```

Notice that, unlike the line item example in a previous section (see “[Exporting Line Items as Separate Records](#)” on page 414), no blank lines are exported. Panorama counts the number of elements in the array, and outputs exactly that number of lines. If you use multiple `arrayscan()` functions in the formula, Panorama will export enough lines to handle the largest array.

The `arrayscan()` function can also be used in the formula for the `arraybuild`, `arrayselectedbuild`, or `arraylinebuild` statements (see “[ARRAYBUILD](#)” on page 5038 of the *Panorama Reference*). The `arrayscan()` works exactly the same as it does with the `export` statement, but the final result is an array instead of a text file.

Opening a Document in Another Application

You can use the `openanything` statement to open any document on your hard drive in its native application. For example, you can use this statement to open a `.doc` file in Microsoft Word, or a `.pdf` file in Adobe Acrobat (or Apple’s Preview program, if that is the default viewer for `.pdf` files on your system).

This example opens the Adobe Acrobat document `Manual.pdf` in the folder `My Documents`.

```
openanything folder("C:\My Documents"), "Manual.pdf"
```

This example opens the HTML document `Roadshow.html` in the folder `My Test Site`. The HTML document will be opened using your default web browser, which is usually Internet Explorer on Windows computers and Safari on Macintosh computers.

```
openanything folder("My Drive:My Test Site"), "Roadshow.html"
```

You can also use `openanything` to directly open an application without opening any document, like this:

```
openanything folder("My Drive:Applications"), "Marine Aquarium.app"
```

This opens the applications just as if you had double clicked on it.

Smart Merge Synchronization

If you have the same database on more than one machine (for example on a desktop computer and a portable computer) you may sometimes wonder which contains the most up-to-date information. If you build **smart merge** into the database, you won't have to wonder anymore. Panorama will keep track of which information is most up-to-date on a record by record database. When you run a special merge procedure, Panorama will merge the two databases, picking the most up-to-date information from each. It's quite easy to add this feature to almost any Panorama database.

(Note: If you are using Panorama's multi-user Partner/Server capabilities, you do not need to add separate **smart merge** synchronization—Panorama automatically synchronizes all copies of the database using the server.)

How Smart Merge Synchronization Works

Smart merge works by comparing two database files record by record, keeping only the most recently modified version of each record. To do this it must be able to match up the corresponding records in the two database files, even if the database files have been sorted or otherwise rearranged. To do this it uses a special ID field. This field contains a unique ID value for every record in the database. The unique ID value is assigned when the record is first created, and never changes no matter how many times the record is modified or copied to other computers. To guarantee that the ID value is unique it is created by combining the name of the person creating the record along with a serial number, for example [Lisa267](#) or [John3091](#). Because of this, every computer you plan to use must have a different user name configured.

Adding Smart Merge to Your Database

The first step is to add two new fields to your database. We usually call these fields **ID** and **Modified**. This illustration shows these fields added to the design sheet of an address book database.

Field Name	Type	Di	Align	Out	Inp	Range	Choi	Link	Clair	Tab	Cap	Dup	Def	Equ	Re
Zip	Text	0	Left			AZ 09			Off	1 S	Off	Yes	"		
Country	Text	0	Left			AZ az			Off	2 S	All	Yes	US		
Phone	Text	0	Left			(Numer			Off	Off	Off	No	I		
Fax	Text	0	Left			(Numer			Off	Off	Off	Yes			
Email	Text	0	Left			AZaz0			Off	Off	Off	Yes			
Notes	Text	0	Left			Any			Off	Off	Off	Yes			
Sequence	Nume	0	Right			Any			Off	Off	Off	Yes			
ID	Text	0	Left			Any			Off	Off	Off	Yes			
Modified	Nume	0	Right			Any			Off	Off	Off	Yes			

The **ID** field will contain the unique ID value for each record, and should be a text field. The **Modified** value will contain the most recent modification date and time for each record, and should be a numeric field. You may also want to have a creation date field, but this is not necessary for the operation of the smart merge feature.

The Modified Field

Panorama can automatically update the **Modified** field with the current date and time whenever the database is modified. This feature is called time stamping. To enable this feature, open the design sheet for your database. Choose the **Time Stamp Field** command from the Special menu (see “[Automatic Time/Date Stamping](#)” on page 301 of the *Panorama Handbook*). This opens a dialog box with a pop-up menu. The pop-up menu lists all the integer fields (Numeric 0 digits) in your database — use the menu to select the **Modified** field.



(If you don't see the **Modified** field listed, close the dialog, press the **New Generation** tool, then open the **Time Stamp Field** dialog again.)

Once you have designated the **Modified** field as the time stamp field, Panorama will automatically place a SuperDate containing the latest date and time into the field every time a new record is added, or whenever any other cell in the record is modified (see “[SuperDates \(combined date and time\)](#)” on page 118).

Adding New Records

Whenever a new record is added to your database, you must make sure that the **ID** field is filled in. The best way to do this is to add a **.NewRecord** automatic procedure to your database (see “[.NewRecord](#)” on page 386). The line shown below will fill in the proper value in the **ID** field.

```
ID=uniqueid("ID",info("user"))
```

Although you may find other uses for it, the **uniqueid()** function was designed specifically for creating unique smart merge serial numbers (see “[UNIQUEID\(\)](#)” on page 5870 of the *Panorama Reference*). This function has two parameters: the name of the field containing the ID serial numbers and a root name. You can get the root name by using the **info("user")** function. The **uniqueid()** function will scan the **ID** field to find the next serial number available. For example, if you are using a computer belonging to **Sam** and the highest **Sam** serial number is **296**, the **uniqueid()** function will return the value **Sam297**.

Creating an **.NewRecord** procedure may not be enough to insure that the **ID** field is always filled in. If your database has procedures that create new records, the **.NewRecord** procedure will not automatically be called. You must modify these procedures to call the **.NewRecord** procedure (using the **call** statement).

Another possible problem area is imported data. When you import data into the database you must make sure that the **ID** and **Modified** fields are filled in. The procedure listed below will do the job. You should also run this procedure when you first add smart merge to your database, so that all your existing data will be properly identified.

```
select ID=""
field Modified
formulafill superdate(today(),now())
field ID
formulafill uniqueid("ID",info("user"))
selectall
```

When you first run this procedure after adding the **ID** and **Modified** fields it will initialize the fields something like this.

First	Last	Credit Card	Title	ID	Modified
Keith	Baker		Sales Manage	Joe Smith1	-1243807620
Nabil	Basir			Joe Smith2	-1243807620
John	Bath		President	Joe Smith3	-1243807620
Jack	Beardsley		Sales Manage	Joe Smith4	-1243807620
Carl	Berg		Owner	Joe Smith5	-1243807620
Leslie	Bianchi			Joe Smith6	-1243807620
Mary	Bilbury		Vice Presiden	Joe Smith7	-1243807620
Joseph	Bizzarri		Owner	Joe Smith8	-1243807620
David	Blair		Owner	Joe Smith9	-1243807620
Al	Bodner			Joe Smith10	-1243807620

The Smart Merge Procedure

Once the **ID** and **ModDate** fields are set up, you're ready to build the actual smart merge procedure. This procedure performs three basic operations.

1. Append second file to the current file.
2. Sort by ModDate within ID.
3. Remove records with duplicate ID's

The first step is to append the file you want to merge with the current file. A procedure can do this with the **openfile** command by putting a plus sign in front of the file name, for example, **openfile "+"MergeFile** (see "[Appending Databases End-to-End](#)" on page 406). The big question is, what file do you want to merge with? If you know the filename in advance, you can simply enter the name into the procedure itself. For example, if you always want to merge with a file named **Invoices** on a floppy named **Data**, the procedure should contain the statement

```
openfile "+Data:Invoices"
```

Usually you'll want to let the user choose what file he or she wants to merge with. The example below shows how this can be done with the **openfiledialog** statement (see "[Locating a File with Standard Dialogs](#)" on page 402).

The next step is to sort the database (see "[Sorting](#)" on page 551). The database must be sorted by both the **Modified** field and the **ID** field. If two records have the same ID value they will now be right next to each other, with the more up-to-date record closer to the bottom.

The final step is to remove the duplicates (see “[Using UnPropagate to Eliminate Duplicates](#)” on page 584). The `unpropagateup` statement identifies the duplicate records (see “[UnPropagate](#)” on page 469). If the same `ID` value occurs more than once in a row, this statement will clear all but the last value. Once the duplicates are identified, the `select` and `removeunselected` statements delete the duplicates from the file.

```

local PathFile,mergeFile, mergePath,mergeType, DoneID
openfiledialog mergePath,mergeFile,mergeType,"ZEPD"
if mergeFile=""
    stop /* user pressed cancel */
endif
PathFile=pathstr(mergePath)+mergeFile
if mergeFile<>info("databasename")
    alert 1014,"Are you sure you want to merge "+
    mergeFile+" with "+info("databasename")+ "?"
    if info("dialogtrigger") contains "no"
        stop
    endif
endif
DoneID=ID ; so we can go back to this record when finished
openfile "+"+PathFile
noshow
field Modified
sortup
field ID
sortup
unpropagateup
select ID<>""
removeunselected
field SortName
sortup
find ID=DoneID
showpage
endnoshow

```

This **Smart Merge** procedure will work for any database that has the `ID` and `Modified` fields properly set up. To use this procedure simply select **Smart Merge** from the **Action** menu.

Directly Reading and Writing Disk Files

Disk files are used for permanent storage of information. Panorama normally takes care of saving information in disk files, and reading it back in again later as needed. However Panorama also gives you the flexibility of accessing disk files directly.

Your disk may contain hundreds or thousands of files. When you create a new file, the operating system (MacOS or Windows) allocates a space on the disk for it. As the file grows, more space is made available as needed (until the disk is full). The operating system keeps track of the exact location and size of each file for you, and makes sure that each file is kept separately and doesn't interfere or overlap with any other file.

What's in a File?

Before launching into the actual business of reading and writing files a little background is in order. Different types of files contain different types of information. A particular file may contain a program, a picture, text, a database a spreadsheet, etc. When reading and writing files it's often important to know what kind of file it is and what it is supposed to contain. On Windows PC systems a file's type can be surmised from the three or four letter extension at the end of the file name — **.exe** for programs, **.txt** for text files, etc. On the Macintosh file extensions are not used for this purpose. Instead, each file has two invisible **tags** that identify what type of file is, and how the file was created. Each invisible tag is four characters long, for example **TEXT**, **APPL**, or **KASX**. You cannot normally see or modify these invisible tags, but you can access them via program statements and functions.

There are literally thousands of different tags and extensions for identifying different types of files. On the Macintosh the tag that identifies the type of file is called the **file type tag**. This table lists a few of the extensions and corresponding file type tags that you may encounter. (Note: Some of these tags, for example PDF, are only three characters long - in that case a space must be added to the end.)

Extension (PC)	Tag (Mac)	File Contents
.exe	APPL	Application (program)
.txt	TEXT	Text file
.pan	ZEPD	Panorama Database
.pnz	KSET	Panorama File Set
.pwp	paig	Panorama Word Processing document (see " Word Processor Document Storage Strategies " on page 696 of the <i>Panorama Handbook</i>)
.pct	PICT	Macintosh PICT graphic (image)
.png	PNGf	Portable Network Graphic (image)
.jpg	JPEG	JPEG image
.tif	TIFF	TIFF image
.eps	EPSF	Encapsulated Postscript image (EPS)
.pdf	PDF	Adobe Acrobat Document
.mov	MOOV	Quicktime Movie
.wav	WAVE	Sound file
.aif	AIFF	Sound file
.sit	SITD	Stuffit Archive (compressed file)

Files on a Macintosh also include an additional four character tag that identifies what application created the file. This is called the **file creator tag**. Windows doesn't really have a corresponding information. The computer uses this tag to decide what application to launch when you double click on the file. Here are a few of the file creator tags you may encounter.

Tag	Application
KASX	Panorama (3.5 or later)
KAS1	Panorama (3.1 or earlier)
ttxt	SimpleText (text editor)
R*ch	BBEdit (text editor)
CWIE	CodeWarrior (text editor)
MPS	Macintosh Programmers Workshop
ToyS	AppleScript Editor
8BIM	Adobe Photoshop
XCEL	Microsoft Excel
WDBN	Microsoft Word
SIT!	Stuffit Archive

Many Panorama functions use the type and creator tags to select files to be displayed or processed.

Reading Data Files

Panorama has three functions for getting information from data files: `fileload()`, `fileloadpartial()`, and `filesize()`.

The `fileload()` function reads an entire file. The data in the file can be copied directly into a field or variable, or processed further using the formula. The function has two parameters: a folder ID (see "[Folder ID's and Paths](#)" on page 401) and a file name. This example loads the contents of the system `Note Pad` into a field or variable named `Notes`.

```
local Notes
Notes=fileload(info("systemfolder"),"Note Pad File")
```

The `fileload()` function can read the data in any file on your disk. It's up to you to interpret what the data means, however—the `fileload()` function simply reads the raw data, exactly as it appears on the disk. Many files (if not most) will contain unintelligible gobbledygook.

If Panorama cannot read the file because of an error, the function will result in an error. In a procedure, this error can be trapped with the `if error` statement (see "[Error Handling with if error](#)" on page 258).

The `fileloadpartial()` function is similar to `fileload()`, but it reads only a section of the file. It has two additional parameters, the starting and ending positions within the file. These positions are measured in characters from the beginning of the file, with 0 being the first character. The example below reads the first 500 characters from the file `My Data` in the current folder, then extracts the first line from the data.

```
local LineOne
LineOne=array(fileloadpartial("", "My Data", 0, 500), 1, ¶)
```

This same example of extracting the first line would also work with the `fileload()` function, but only if there is enough scratch memory to load the entire file. By using `fileloadpartial()` this procedure requires only 500 bytes of scratch memory no matter how large the file `My Data` is.

The `filesize()` function calculates the size of a file. It has two parameters, the folder id (see “[Folder ID's and Paths](#)” on page 401) and file name.

```
if filesize("", "Sample File")=0
  message "The file is empty!"
endif
```

Note: If a file named `Sample File` does not exist in the current folder, the procedure above will display the error message `File not found`. Use the `if error` statement to trap this error if you want to display your own error or handle the error differently (see “[Error Handling with if error](#)” on page 258).

```
local size
size=filesize("", "Sample File")
if error
  message "Sample File does not exist!"
  rtn
endif
message "Sample File contains "+str(size)+" bytes."
```

Reading Really Big Data Files

The `fileload()` function works fine for files up to about 2 to 3 megabytes in size. When you go beyond that you may find that you need to expand the “expression stack.” The **expression stack** is a section of memory that Panorama reserves for processing data within formulas. Panorama normally allocates 5 megabytes of memory for the expression stack, which is far more than is needed for ordinary numeric and text calculations. If you are directly working with large files, however, your needs can easily exceed this limit.

To change the size of the expression stack use the `expressionstacksize` statement.

```
expressionstacksize bytes
```

You can use the `expressionstacksize` statement to increase the available memory up to 250 megabytes. Keep in mind, however, that increasing this allocation reduces the amount of memory available for other applications. The new memory allocation is semi-permanent, remaining in effect until you change it again or until Panorama quits. You can find out the current allocation using the `info("expressionstacksize")` function. (Keep in mind that although the `expressionstacksize` statement allows you to manipulate large amounts of data you cannot store more than 4 megabytes of data in a Panorama data cell.)

This example allows the `myFile` variable to be loaded with a very large amount of data. If the expression stack isn't large enough, it is expanded automatically. However, if there is not enough memory available on the system then an error message is displayed.

```
local myFile, myFileSize
myFileSize= filesize( "", "MyPicture.jpg" )
if myFileSize > info("expressionstacksize" ) * 2
  expressionstacksize myFileSize * 2
  if error
    message info("error")
    rtn
  endif
endif
myFile= fileload( "", "MyPicture.jpg" )
```

Another method for handling large amounts of data is to use the `extendedexpressionstack` statement. This statement allows a formula to use free database memory for formula calculations. For example, if you have 50 megabytes free, you'll be able to use all 50 megabytes for your formula data. When you are done with whatever operations require the extra large amount of memory you should use the `normalexpressionstack` statement.

Unlike the `expressionstacksize` statement, the `extendedexpressionstack` statement is temporary: it applies to formulas until you use the `normalexpressionstack` statement or until you move to a different record, move to a different database, edit a cell, use a fill command, or use the clipboard. In general we recommend that you use the `normalexpressionstack` statement as soon as possible after the `extendedexpressionstack` statement. Another difference is that the `extendedexpressionstack` statement does not apply to formulas that are in procedures that run as handlers, or to formulas that are embedded in form objects, while the `expressionstacksize` statement. For most applications the `expressionstacksize` statement should be the first choice.

This example uses the `extendedexpressionstack` statement to allow the `myFile` variable to be loaded with a very large amount of data. Keep in mind that if you later want to access this variable you'll need to use the `extendedexpressionstack` statement again.

```
local myFile
extendedexpressionstack
myFile= fileload( "", "MyPicture.jpg" )
normalexpressionstack
```

Remember, none of this applies unless you are working with data that is more than 2 or 3 megabytes in size.

Writing Data Files





The `filesave` statement copies data into a file. If the file does not already exist, it is created. If the file already contained information, that information is lost.

The `filesave` statement has four parameters

```
filesave folder,file,type,data
```

The first parameter, `folder`, is the folder ID where the file is to be saved. The second parameter is the name of the file.

The third parameter, `type`, is an 8 character text item combining the **file type tag** and **file creator tag** for the file (see “[What’s in a File?](#)” on page 421). If you are using a Windows PC you can simply use "" for this parameter. If you are using a Macintosh the type and creator tags determine what icon, if any, this file will have, and what application will be launched when you double click on this file. If you use an empty string ("") for this parameter, the file will be set up as a **SimpleText** text file. Although you can create any type of file the most common application is to create a text file, as shown in this table.

Icon	Type/Creator	Description
	TEXTttxt	SimpleText text file
	TEXTR*ch	BBEdit text file
	TEXTCWIE	CodeWarrior text file
	TEXTMPW	MPW text file

The fourth parameter, `data`, is a formula that produces the data to be saved into the file. This may be a field, variable, or more complex formula.

Here is an example that saves the contents of the **Notes** field (the current record only) into a text file called **Notes.txt**.

```
filesave "", "Notes.txt", "TEXT*ch", Notes
```

The **fileappend** statement adds data to the end of an existing file. If the file does not already exist, it is created. If the file already contained information, that information is retained and the new information is added to the end of the file. The procedure below adds a line to the end of the file **Deleted Records Log.txt** every time it is triggered.

```
fileappend "", "Deleted Records Log.txt", "", "Record "+str(ID)+" deleted on "+
  datepattern(today(), "Month, ddnth, yyyy")+ " at "+timepattern(now(), "hh:mm:ss am/pm")+
  deleterecord
```

If this procedure is named **.DeleteRecord** it will be triggered every time a record is deleted (see “**.DeleteRecord**” on page 381). As records are deleted a log file will be created that looks something like this.

```
Record 4738 deleted on June 4th, 2001 at 3:12:47 PM
Record 392 deleted on June 4th, 2001 at 4:45:02 PM
Record 6133 deleted on June 5th, 2001 at 9:23:20 AM
```

As more records are deleted the log file will get bigger and bigger and bigger.

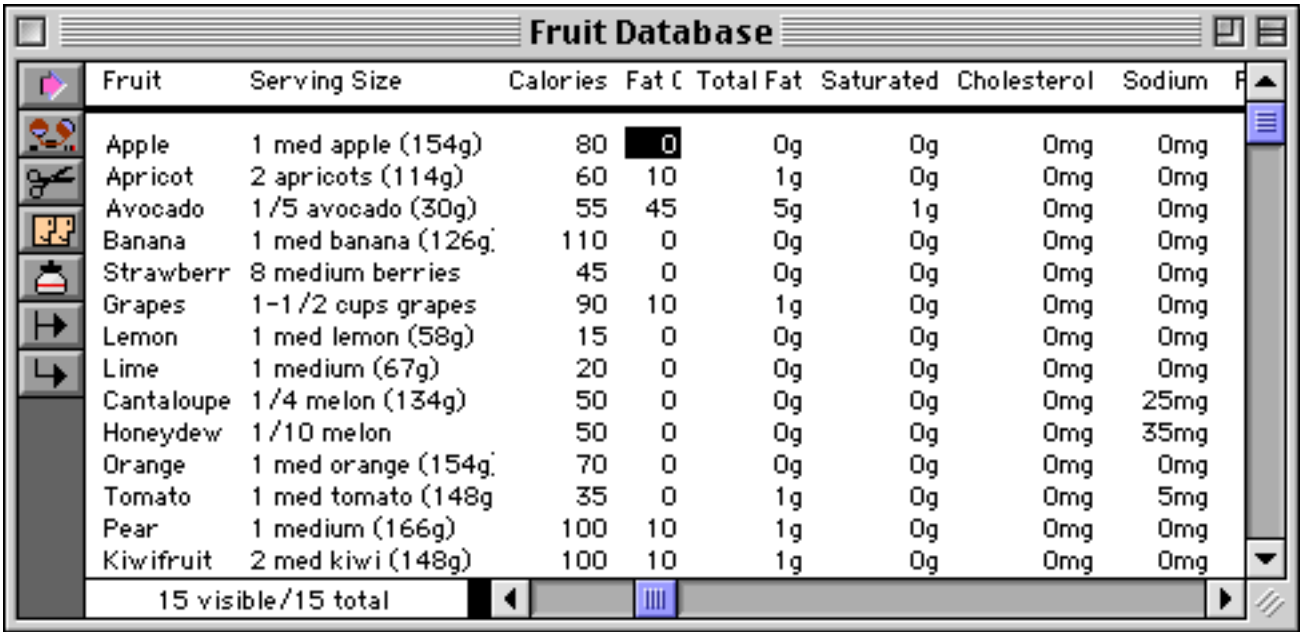
Copying Data Files

Panorama has several built in statements for copying files or parts of files. To learn more about these commands see the **copyfile** and **copyfork** statements in the **Programming Reference** wizard.

Using FileSave and ArrayBuild to Export Data

Earlier in this chapter we learned how to use the **export** statement to export data from the database (see “**Exporting Text Files**” on page 413). Another technique is to use the **arraybuild** statement to scan the database and build an array (see “**ARRAYBUILD**” on page 5038 of the *Panorama Reference*) and then use **filesave** to export the array into a file. The advantage of this technique is that it is more flexible than using **export** because it allows you to add headers and footers to the data. When using Mac OS 9 the disadvantage is that this technique requires at least enough scratch memory to contain the entire exported file, so it will not work for exporting large databases. On Windows and Mac OS X Panorama can take advantage of virtual memory for this, so it is less of a problem.

To illustrate this technique we will use it to export a file as an HTML table. We’ll start with a database of fruit nutrition.



Fruit	Serving Size	Calories	Fat (g)	Total Fat (g)	Saturated (g)	Cholesterol (mg)	Sodium (mg)
Apple	1 med apple (154g)	80	0	0g	0g	0mg	0mg
Apricot	2 apricots (114g)	60	10	1g	0g	0mg	0mg
Avocado	1/5 avocado (30g)	55	45	5g	1g	0mg	0mg
Banana	1 med banana (126g)	110	0	0g	0g	0mg	0mg
Strawberry	8 medium berries	45	0	0g	0g	0mg	0mg
Grapes	1-1/2 cups grapes	90	10	1g	0g	0mg	0mg
Lemon	1 med lemon (58g)	15	0	0g	0g	0mg	0mg
Lime	1 medium (67g)	20	0	0g	0g	0mg	0mg
Cantaloupe	1/4 melon (134g)	50	0	0g	0g	0mg	25mg
Honeydew	1/10 melon	50	0	0g	0g	0mg	35mg
Orange	1 med orange (154g)	70	0	0g	0g	0mg	0mg
Tomato	1 med tomato (148g)	35	0	1g	0g	0mg	5mg
Pear	1 medium (166g)	100	10	1g	0g	0mg	0mg
Kiwifruit	2 med kiwi (148g)	100	10	1g	0g	0mg	0mg

Here is the procedure that takes this database and creates an HTML page. The heart of the procedure is the `arraybuild` statement, which scans the database and creates each line of the table. See “[ARRAYBUILD](#)” on page 5038 of the *Panorama Reference* to learn about the parameters to this statement.

```

local webPage,webTable

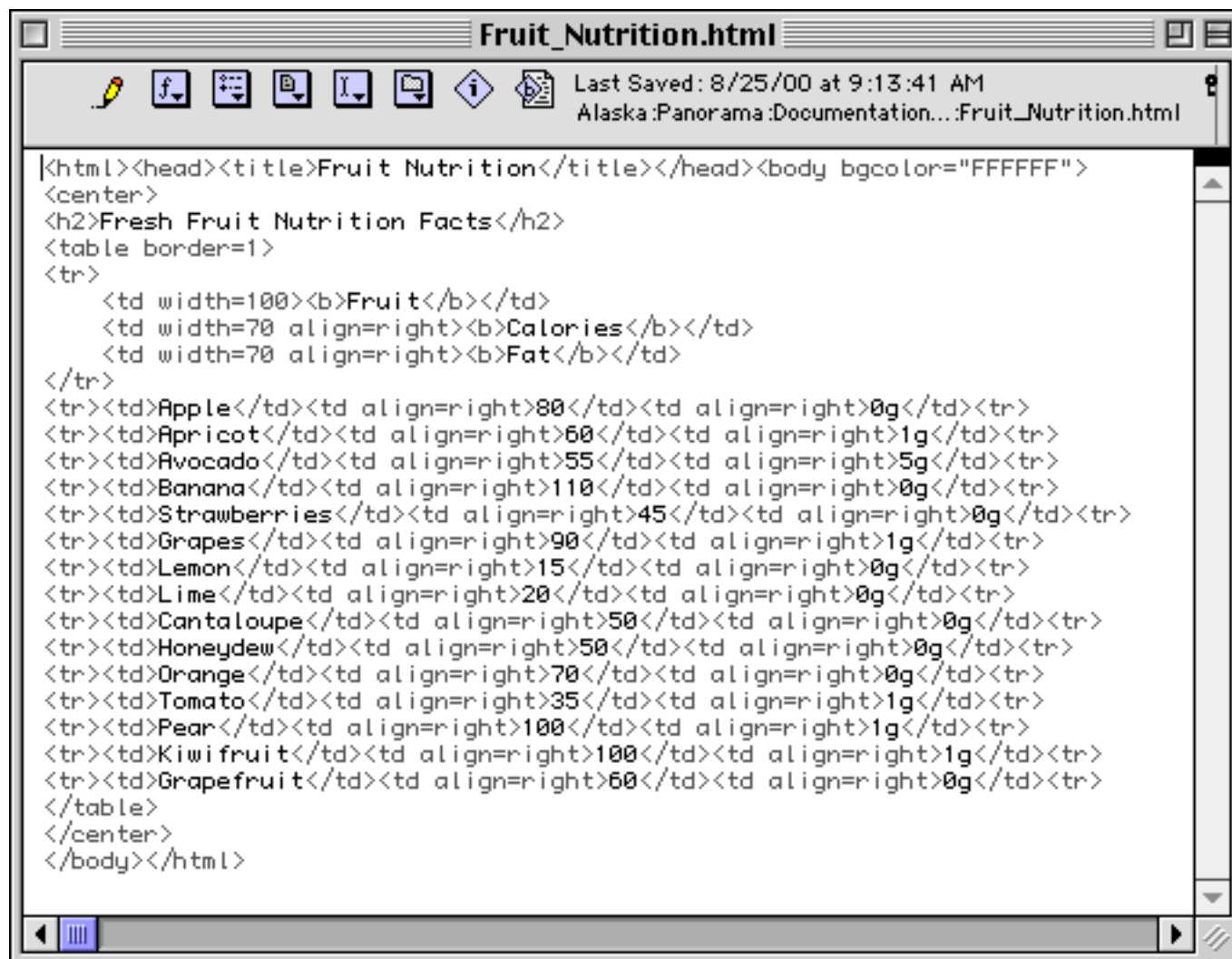
arraybuild webTable,¶,"",
  "<tr><td>"+Fruit+"</td>"+
  "<td align=right>"+str(Calories)+"</td>"+
  "<td align=right>"+str(«Total Fat»)+"g</td><tr>"

webPage={<html><head><title>Fruit Nutrition</title></head><body bgcolor="FFFFFF">
<center>
<h2>Fresh Fruit Nutrition Facts</h2>
<table border=1>
<tr>
  <td width=100><b>Fruit</b></td>
  <td width=70 align=right><b>Calories</b></td>
  <td width=70 align=right><b>Fat</b></td>
</tr>
<DATA>
</table>
</center>
</body></html>}

filesave "", "Fruit_Nutrition.html", "TEXT*ch", replace(webPage, "<DATA>", webTable)

```

The middle section of the procedure places a web page template into the variable `webPage`. Notice how the `{` and `}` characters are used around the text so that `"` may be used within the text (see “[Constants](#)” on page 49). The final line uses the `replace()` function to merge the table body into the web page template and then saves the file, which will look something like this.

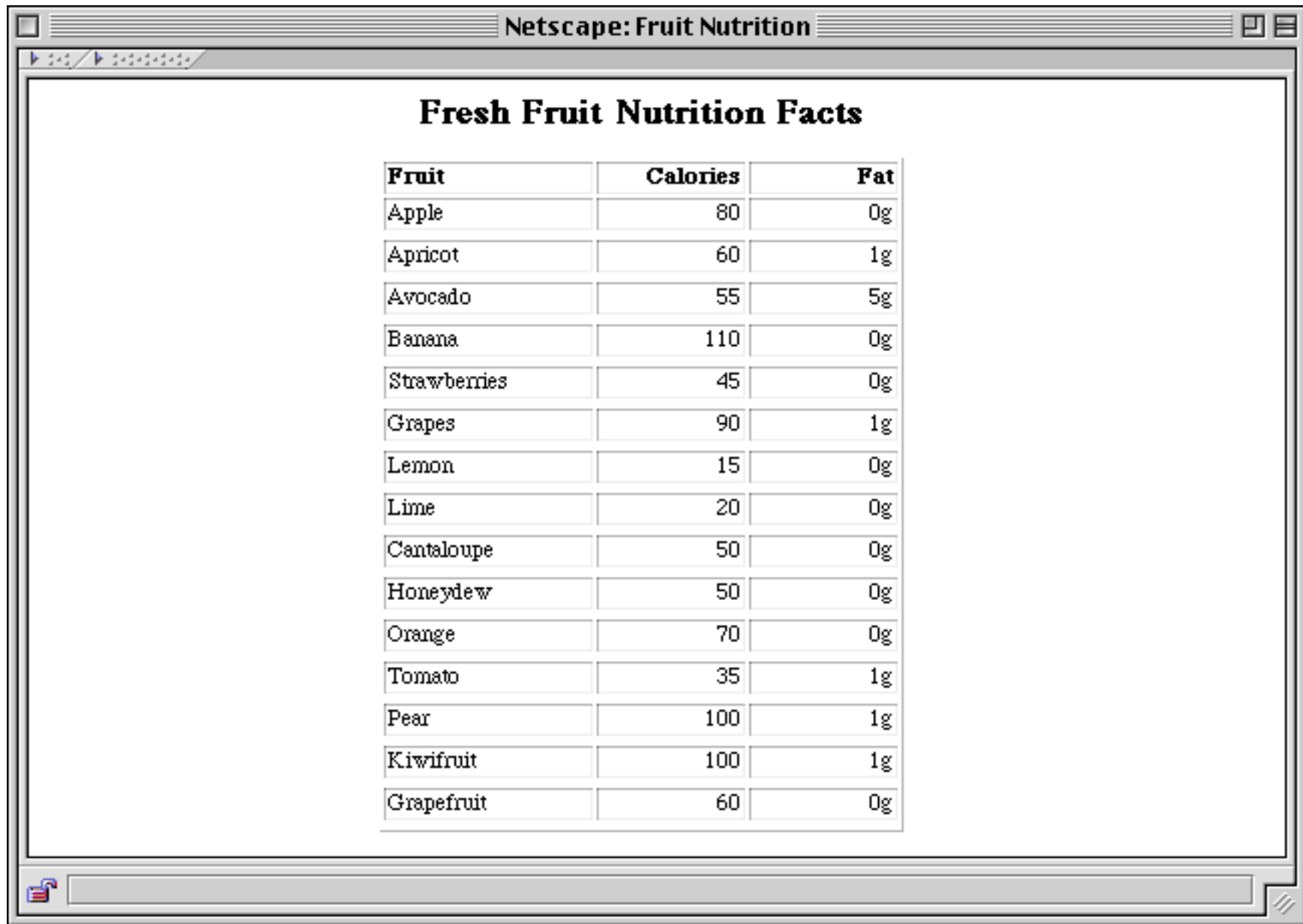


```

<html><head><title>Fruit Nutrition</title></head><body bgcolor="FFFFFF">
<center>
<h2>Fresh Fruit Nutrition Facts</h2>
<table border=1>
<tr>
  <td width=100><b>Fruit</b></td>
  <td width=70 align=right><b>Calories</b></td>
  <td width=70 align=right><b>Fat</b></td>
</tr>
<tr>
<td>Apple</td><td align=right>80</td><td align=right>0g</td><tr>
<tr><td>Apricot</td><td align=right>60</td><td align=right>1g</td><tr>
<tr><td>Avocado</td><td align=right>55</td><td align=right>5g</td><tr>
<tr><td>Banana</td><td align=right>110</td><td align=right>0g</td><tr>
<tr><td>Strawberries</td><td align=right>45</td><td align=right>0g</td><tr>
<tr><td>Grapes</td><td align=right>90</td><td align=right>1g</td><tr>
<tr><td>Lemon</td><td align=right>15</td><td align=right>0g</td><tr>
<tr><td>Lime</td><td align=right>20</td><td align=right>0g</td><tr>
<tr><td>Cantaloupe</td><td align=right>50</td><td align=right>0g</td><tr>
<tr><td>Honeydew</td><td align=right>50</td><td align=right>0g</td><tr>
<tr><td>Orange</td><td align=right>70</td><td align=right>0g</td><tr>
<tr><td>Tomato</td><td align=right>35</td><td align=right>1g</td><tr>
<tr><td>Pear</td><td align=right>100</td><td align=right>1g</td><tr>
<tr><td>Kiwi fruit</td><td align=right>100</td><td align=right>1g</td><tr>
<tr><td>Grapefruit</td><td align=right>60</td><td align=right>0g</td><tr>
</table>
</center>
</body></html>

```

When displayed in a web browser the finished result looks like this.



The screenshot shows a Netscape browser window with the title "Netscape: Fruit Nutrition". The main content area displays a table titled "Fresh Fruit Nutrition Facts". The table has three columns: "Fruit", "Calories", and "Fat". The data is as follows:

Fruit	Calories	Fat
Apple	80	0g
Apricot	60	1g
Avocado	55	5g
Banana	110	0g
Strawberries	45	0g
Grapes	90	1g
Lemon	15	0g
Lime	20	0g
Cantaloupe	50	0g
Honeydew	50	0g
Orange	70	0g
Tomato	35	1g
Pear	100	1g
Kiwifruit	100	1g
Grapefruit	60	0g

This example used the `arraybuild` statement to export the entire file. If you wanted to export only the currently selected records you would need to use the `arrayselectedbuild` statement.

Reading and Writing Resource Forks

On the Macintosh each file may consist of two separate partitions called **forks**, the **data fork** and the **resource fork**. The resource fork is normally accessed only indirectly through special statements (see “[Working with Resources](#)” on page 433) and not through the standard file i/o statements and functions. However, sometimes it is necessary to read and write the resource fork directly (for example to copy a file you must copy both forks). To do this you must use the **resourcefork** statement to switch Panorama into resource fork mode (see “[RESOURCEFORK](#)” on page 5668 of the *Panorama Reference*). In this mode all of Panorama’s normal file i/o statements and functions access the resource fork instead of the data fork. To go back to normal data fork mode use the **datafork** statement (see “[DATAFORK](#)” on page 5141 of the *Panorama Reference*).

Here is a procedure that makes a copy of a file named **My File**. The copy is called **Copy of My File**, and includes both the data and resource forks from the original file.

```
local theOriginalFile,typecreator,data

theOriginalFile="My File"

typecreator=array(fileinfo("",theOriginalFile,2,¶))

datafork
data=fileload("",theOriginalFile)
filesave "", "Copy of "+theOriginalFile,typecreator,data

resourcefork
data=fileload("",theOriginalFile)
filesave "", "Copy of "+theOriginalFile,typecreator,data

datafork
```

On Windows PC systems files do not have resource forks and the **resourcefork** statement does absolutely nothing. If you want your program to work on both Mac and PC systems you must check which system you are using and only copy the resource fork if the database is on a Macintosh. Here is a revised copy of the procedure which shows one way to perform this check.

```
local theOriginalFile,typecreator,data,computerType

computerType="Macintosh"
if folderpath(dbinfo("folder","")) match "?:\*"
  computerType="Windows"
endif

theOriginalFile="My File"

typecreator=array(fileinfo("",theOriginalFile,2,¶))

datafork
data=fileload("",theOriginalFile)
filesave "", "Copy of "+theOriginalFile,typecreator,data

if computerType="Macintosh"
  resourcefork
  data=fileload("",theOriginalFile)
  filesave "", "Copy of "+theOriginalFile,typecreator,data
  datafork
endif
```

Erasing a File

A procedure can erase any file by using the `filetrash` statement. This statement has two parameters: the `folder ID` (see “[Folder ID’s and Paths](#)” on page 401) and the `name` of the file to be erased. (If you use "" as the folder ID, the folder containing the current database is assumed.) This procedure will erase the file `Temp Data File.txt` in the System folder.

```
filetrash info("systemfolder"),"Temp Data File.txt"
```

Keep in mind that the `filetrash` statement is the same as dragging the file into the trash can or recycle bin, then choosing the `Empty Trash` or `Empty Recycle Bin` command. Once a file has been deleted with this statement, you cannot get it back, so be careful.

Changing a File’s Name

A procedure can change the name of any file by using the `filerename` statement. This statement has three parameters: the `folder ID` (see “[Folder ID’s and Paths](#)” on page 401), the `original name` of the file, and the `new name` of the file. (If you use "" as the `folder ID`, the folder containing the current database is assumed.) This procedure will rename the file `Temp Data File.txt` in the same folder as the current database, giving it the new name `Permanent Data File.txt`.

```
filerename "", "Temp Data File.txt", "Permanent Data File.txt"
```

If there is already a file named `Permanent Data File.txt`, this statement will not be able to rename the file.

Changing a File’s Type and Creator

On the Macintosh each file has two four character tags that identify what type of file it is and what application it belongs to (see “[What’s in a File?](#)” on page 421). A procedure can change these tags by using the `filetypecreator` statement (see “[FILETYPECREATOR](#)” on page 5240 of the *Panorama Reference*). This statement has three parameters: the `folder ID` (see “[Folder ID’s and Paths](#)” on page 401), the `name` of the file, and the `new tags` for the file. (If you use "" as the `folder ID`, the folder containing the current database is assumed.)

The procedure below examines a file that has been transferred from a PC computer to a Macintosh. Depending on the three character "extension" at the end of the filename, it converts the file into a text file, a Panorama file, or a Photoshop picture file.

```
case myfile endswith ".txt"
  filetypecreator myfolder,myfile,"TEXTttxt"
case myfile endswith ".pan"
  filetypecreator myfolder,myfile,"ZEPDKASX"
case myfile endswith ".pct"
  filetypecreator myfolder,myfile,"PICT8BIM"
endcase
```

Since PC files do not have type/creator tags the `filetypecreator` statement is simply ignored when used on a Windows PC.

Creating a New Folder

If it is necessary for a procedure to create a new folder it can do so with the `makenewfolder` statement. This statement has one parameter, the full path of the new folder. Here is an example. This example creates a folder named `Project Foo`.

```
makenewfolder "C:\Plans\1999\Project Foo"
```

If you want to check whether or not a folder already exists you can use the `folderexists()` function.

Getting Information about a File

Your computer keeps track of a number of attributes for each file on the disk, including the size of the file, the time and date when it was created, the time and date when it was last modified, and (on the Macintosh) the file type tag and file creator tag. Panorama has several functions for accessing this information: `fileexists()`, `filedate()`, `filetime()`, `filetypecreator()` and `fileinfo()`. Each of these functions have two parameters: the **folder ID** (see “[Folder ID’s and Paths](#)” on page 401) and the **filename**.

The `fileexists()` function returns true or false depending on whether the file exists or not.

The `filedate()` function returns the date when this file was last modified (see “[Date Arithmetic](#)” on page 106).

The `filetime()` function returns the time when this file was last modified. This time is the number of seconds after midnight (see “[Time Arithmetic](#)” on page 113).

The `filetypecreator()` function returns the 8 characters defining the **type tag** and **creator tag** for the file (4 characters each - see “[What’s in a File?](#)” on page 421). You can use this information to determine the type of file. For example, a Panorama database is **ZEPDKASX**. On PC systems Panorama will attempt to simulate the type and creator tags for some types of files based on the file’s extension (.txt, .pan, etc.)

The `fileinfo()` function returns a text array with 8 items of information. The eight items are combined together in an array with carriage return separators, so you can use the `array()` function to extract the information you want.

The first element in the array returned by `fileinfo()` is the **type of item**, which may be either **File** or **Folder**.

The second element in the array returned by `fileinfo()` is 8 characters defining the **type tag** and **creator tag** for the file (4 characters each - see “[What’s in a File?](#)” on page 421). You can use this information to determine the type of file. For example, a Panorama database is **ZEPDKASX**. On PC systems Panorama will attempt to simulate the type and creator tags for some types of files based on the file’s extension (.txt, .pan, etc.)

The third element in the array returned by `fileinfo()` is a number representing the **creation date** of the file. The formula below displays the creation date of the file named **Sample**.

```
datepattern(val(array(fileinfo("", "Sample"), 3, ¶)), "Month ddnth YYYY")
```

The fourth element in the array returned by `fileinfo()` is a number representing the creation time of the file. The formula below displays the creation time of the file named **Sample**.

```
timepattern(val(array(fileinfo("", "Sample"), 4, ¶)), "HH:MM:SS AM/PM")
```

The fifth element in the array returned by `fileinfo()` is a number representing the modification date of the file. The sixth element in the array returned by `fileinfo()` is a number representing the modification time of the file.

The seventh element in the array returned by `fileinfo()` is the size of the file.

The eighth element in the array returned by `fileinfo()` is the status of the file, either **Locked** or **Unlocked**.

Here is a typical array returned by the `fileinfo()` function for the file Panorama (the application itself).

```
File<           Type of item (File or Folder)
APPLKASX<      File type tag (APPL) and file creator tag (KASX)
2450070<       Creation date. Use val( function to convert to number.
54233<        Creation time. Use val( function to convert to number.
2450075<       Modification date. Use val( function to convert to number.
71028<        Modification time. Use val( function to convert to number.
1092657<      File size (just over 1 megabyte)
Unlocked<     File options (Locked or Unlocked)
```

Here is another example of information for a Panorama database file:

```
File<          Type of item (File or Folder)
ZEPDKASX<     File type tag (APPL) and file creator tag (KASX)
2450035<      Creation date. Use val( function to convert to number.
401<         Creation time. Use val( function to convert to number.
2450035<      Modification date. Use val( function to convert to number.
923<         Modification time. Use val( function to convert to number.
2320<        File size (just over 1 megabyte)
Unlocked<     File options (Locked or Unlocked)
```

Getting and Setting Additional File Information

The `getfilefinderinfo` statement retrieves a collection of information about a file, including when it was created and last modified and its position within the window.

```
getfilefinderinfo folderID,filename,type/creator,position,flags,creationdate,moddate
```

The first two parameters, `folderID` and `filename`, tell Panorama which file you are interested in. See “[Folder ID’s and Paths](#)” on page 401 for more information about folder ID’s.

The next five parameters are all filled in by the statement. You should supply variables for each of these values. `Type/Creator` is the two four character tags that identify what type of file this is. See “[What’s in a File?](#)” on page 421 for more information about these tags. `Position` is the visual x-y position of this file within the folder (see “[Points](#)” on page 147). `Flags` contain a number of operating system specific options for this file. If bit 14 of this value is set then the file is invisible. `CreationDate` and `ModDate` contain the creation date/time and modification date/time of the file. Both of these values are SuperDates (see “[SuperDates \(combined date and time\)](#)” on page 118).

The `setfilefinderinfo` statement modifies a collection of information about a file, including when it was created and last modified and its position within the window.

```
setfilefinderinfo folderID,filename,type/creator,position,flags,creationdate,moddate
```

The first two parameters, `folderID` and `filename`, tell Panorama which file you want to modify. See “[Folder ID’s and Paths](#)” on page 401 for more information about folder ID’s.

The next five parameters specify the new values for each file option. The parameter descriptions are the same as for the `getfilefinderinfo` statement (see above). If you don’t want to change the `type/creator` value you can simply specify `""`. If you don’t want to change the `position`, `flags`, `creationdate` or `moddate` value specify 0. The procedure below sets the creation date/time and modification date/time to 9 am today.

```
setfilefinderinfo "","Sunset.jpg","",0,0,
    superdate(today(),time("9am")),superdate(today(),time("9am"))
```

All of the other file options (`type/creator`, `position` and `flags`) are left undisturbed.

Accessing and Modifying File Permissions

The MacOS X operating system is based on UNIX, and uses UNIX style file permissions. File permissions allow the system administrator to control who can read and write any file or folder. Panorama can access and modify these permissions with the `getfilepermissions` and `setfilepermissions` statements. To learn more about these statements see the [Programming Reference](#) wizard.

Building a List of Files or Folders

The `filecatalog` statement can build a list of files in folders and subfolders. This statement has four parameters: `Path`, `Wildcard`, `TypeCreator` and `Files`.

The `Path` parameter is the path of the primary folder to be cataloged, for example `"Computer:Alpha:Gamma:Zed:Foo"`. The statement will scan this folder, along with any subfolders inside that folder (and any subfolders inside the subfolders, etc.).

The **Wildcard** parameter lets you specify for matching file names that you want to catalog. The `*` character will match any sequence of characters, while the `?` character will match a single character. For example if you wanted to catalog only JPEG images you could use `*.jpg`, while if you wanted to catalog only web pages you could use `*.html`. The default is `*`, which matches all files.

The **TypeCreator** parameter is another method for specifying the types of files you want to catalog. For example, if this parameter is `TEXT????` then only text files will be included in the final list, if this parameter is `????KASX` then only files created by Panorama will be included. The default is `????????`, which matches all files.

The **Files** parameter must be a field or variable. This is where the final list of files will be placed. Each line contains the file path and file name of a single file.

This example will create a list of all JPEG images inside the **My Images** folder (including subfolders), then displays that list in a dialog.

```
filecatalog "My Drive:My Documents:My Images:", "*.jpg", "", jpegArray
displaydata jpegArray
```

The `listfiles()` function builds a list of the files in a single folder. It is not as flexible as the `filecatalog` statement, but since it is a function it can be used to display information in forms. The list is a text array with a carriage return separator between each file name (see “[Text Arrays](#)” on page 93). The `listfiles()` function has two parameters.

```
listfiles(folder, tags)
```

The **folder** parameter is a folder ID (see “[Folder ID's and Paths](#)” on page 401) that identifies what folder you want to list the contents of.

The **tags** parameter specifies what types of files you want to list. Leave this parameter empty if you want to list all files and folders regardless of type.

If you wish, you may use the **tags** parameter to specify one or more types of files to include in the list. Each type of file is specified by an 8 character combination of type and creator tags (see the previous section). For example, to list Panorama database files the tags parameter should be `ZEPDKASX`. You may use the `?` character in the tags parameter when you don't need to match. For example, many different applications can create text files. To list all text files no matter what application created them, use the tags parameter `TEXT????`. You can combine multiple tag specifications to list more than one type of file, for example `ZEPDKASXTEXT????` to list both Panorama databases and text files.

The `listfiles()` function does not normally include folders in the list of files. However, there are two cases where folders will be listed: 1) if the **tags** parameter is empty, or 2) if the **tags** parameter starts with the `f` character (see “[Special Characters](#)” on page 57). For example, to list Panorama databases and folders use the **tags** parameter `fZEPDKASX`. If you want to list only folders without any files, use type creator tags that are not used by any kind of file, for example `fZZZZZZZZ`.

Since `listfiles()` is a function, it can be used in any formula in a procedure, auto-wrap text object, or SuperObject formula. Here is an example formula that lists all the picture files in the same folder as the current database:

```
listfiles(dbinfo("folder", ""), "PICT????")
```

Building a List of Disks (Volumes)

The `info("volumes")` functions creates a list of disks (volumes) that are currently mounted (active). The list is a text array with a carriage return separator between each file name (see “[Text Arrays](#)” on page 93). The example below uses this function to check to see if the **World Facts Reference** CD is currently available.

```
if 0=arraysearch(info("volumes", "World Facts Reference"), 1, ¶)
  message "World Facts Reference CD is not currently available."
endif
```


Working with Resources

On the Macintosh files are split into two partitions, called **forks**. These forks are the **data fork** and the **resource fork**. The data fork corresponds to a normal file as used on other operating systems (Windows, UNIX, etc.) The resource fork, however, is not handled like a normal file. Instead, the operating system further subdivides this fork into components called **resources**. These resources are like miniature “files within a file” and are used to hold objects needed by programs like menus, images, text, templates, and even program code. Instead of accessing the resource fork directly as a file, programs use the operating system to access these components. Each resource component may be anything from a single character to tens of thousands of bytes of information.

Because resources play such an important part in the operation of Macintosh programs (including Panorama) we have created a mechanism by which resources can be used on Windows PC systems as well. Since Windows PC files do not have two forks the resources must be kept in a separate file. This file must have a name ending with the extension **.rsr**, for example **My Menus.rsr**. Unless specified otherwise, all of the functions and statements described in this section work equally well on both the Macintosh and the PC.

Just as a file is identified by its location (folder) and filename, each resource is identified by its **type** and **ID number**. The **type** is a four letter designation that identifies what type of data is stored in that resource. There are hundreds of different types of resources, with more new types being created all the time. However, the most common types were defined by Apple in 1984 and are still in use today. This table describes some of the most common types.

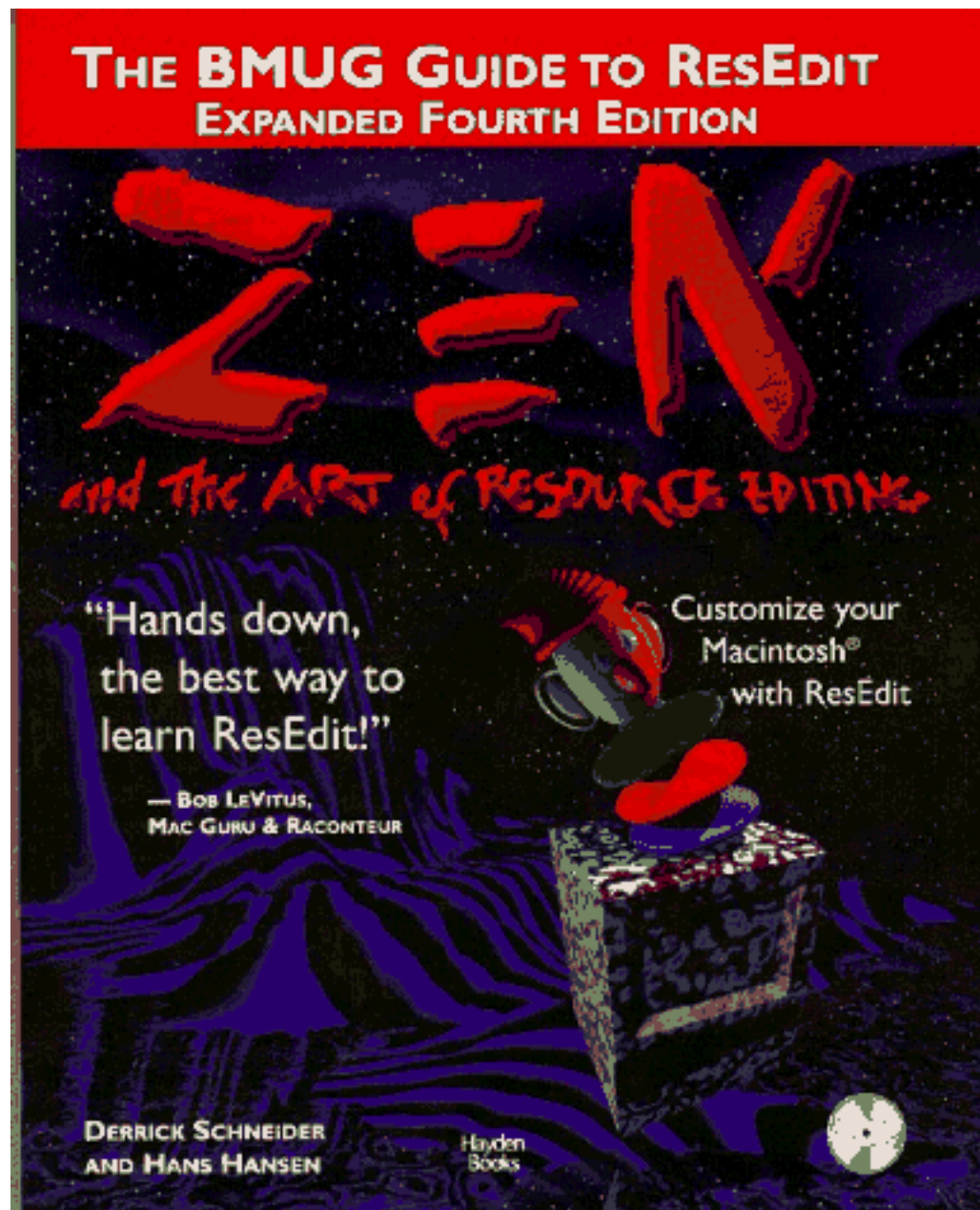
Type	Contents
CODE	Machine code (a program)
MENU	List of items in a menu
STR	A single item of text
STR#	Multiple items of text
DLOG	Template describing a dialog
DITL	List of items within a dialog
PICT	Picture
ICON	A single icon
ICN#	Multiple icons
cicn	Color icon
CURS	Cursor design (mouse pointer)

The **resource ID** is simply a number between 0 and 65535.

Just as a file is identified by its folder and file name, a resource is identified by its type and ID. For example, you may refer to a resource as **MENU 97** or **ICON 2544**.

In addition to a type and ID, a resource may also have a **name**. However, the name is completely optional. If a resource does have a name, you can identify the resource by its type and name as well as by its type and ID. For example you may refer to a resource as **ICON 2544** or as **ICON Empty Trash Can**.

On the Macintosh resource files may be created and edited with resource editing programs. The most popular such program is **ResEdit**, a freeware utility written by Apple and distributed by the Apple Programmers and Developers Association (APDA). ResEdit has appeared in several different versions since the Macintosh was released in 1984. If you'd like to learn more about **ResEdit**, we recommend that you get **Zen and the Art of Resource Editing**, available from many sources, including amazon.com. The book includes a CD with a copy of ResEdit along with many example files.



If you do a lot of resource editing you might want to check out another resource editor, **Resorcerer**, from [Mathemaesthetics](http://www.mathemaesthetics.com/) (<http://www.mathemaesthetics.com/>).



Unlike ResEdit, Resorcerer is not free, but it does have advanced features that are not included in ResEdit. We use Resorcerer instead of ResEdit here at ProVUE.

There are no resource editor programs available for Windows PC's. However, you can still create and modify resources using Panorama statements and functions.

Opening and Closing Resource Files

Before a Panorama procedure can access the objects inside a resource file it must open the resource file. This can be done with the `openresource` or `openresourcerw` statements.

The `openresource` statement opens a resource for read only access — you cannot modify resource objects when a file is opened this way. The `openresource` statement requires one parameter—the name of the resource file to open. For example, if the resource file containing your text is called `Background Items` then the procedure should contain the statement

```
openresource "Background Items"
```

This statement may be used on both Macintosh and Windows PC computers. On Windows PC computers the filename extension of `.rsr` is assumed, so the example above will actually open the resource file `Background Items.rsr` if used on a PC computer.

If the resource file is not in the same folder as the current database you must specify the location as well as the name of the resource file, like this (see “[Combined Folder Location and File Name](#)” on page 400).

```
openresource "C:\Accounting\Background Items"
```

The `openresourcerw` statement also opens a resource file, but allows both reading and writing.

Unlike opening a database window, there is no visible indication when a resource file is opened.

To close a resource file use the `closeresource` statement. The statement must be followed by the name of the resource to close.

```
closeresource "Background Items"
```

If the resource was opened from a different folder you must specify the entire path, like this.

```
closeresource "C:\Accounting\Background Items"
```

It's often not necessary to bother with closing a resource file. If you leave any resource files open Panorama will automatically close them when you exit (**Quit**) from Panorama. If you attempt to re-open a resource file that is already open Panorama simply leaves it open and continues. (However, if a resource is open for reading only you must close it if you want to open it for read/write access.)

It is possible to open more than one resource file at once. In fact, there is always more than one resource file open, because the system has a resource file open, and Panorama has its own resource file open. If two different resource files contain resources with the same type and ID, Panorama will use the copy of the resource from the most recently opened resource file.

To get a list of currently open resource files use the `info("openresourcefiles")` function.

Opening a Resource File in the .Initialize Procedure

If a resource file is required for operation of the database (for example for custom menus), we recommend that you place the `openresource` statement in the `.Initialize` procedure for the file (See “[.Initialize](#)” on page 382 more information on the `.Initialize` procedure).

Simply creating the `.Initialize` procedure does not open the resource file. The first time you create this procedure you must save the database, then close and re-open the database. The `.Initialize` procedure will open the resource file when you re-open the database, and you can begin using the resources immediately. From then on the resource file will be opened automatically every time the database is opened.

Reading a Resource

The `getresource()` function gets a resource from an open resource file and copies it into a field or variable. You can read any resource with this function, although making sense of the contents of the resource is up to you.

The `getresource()` function has two parameters: `type` and `ID`. The `type` is the resource type. This must be a four letter text item (see “[Working with Resources](#)” on page 433). Standard resource types include `"STR "` (Pascal String), `"STR#"` (multiple strings), `"DLOG"` (dialog template), `"DITL"` (dialog items), `"MENU"` (menu). The `ID` is the identification for the resource. The resource ID can be a number (from 0 to 65535) or a name (a text item).

This example loads the contents of TEXT resource number 415 into the field `LetterBody`.

```
openresource "Letter Templates"
LetterBody=getresource("TEXT",415)
```

Remember, all resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example loads the contents of the TEXT resource named `Thank You #2` into the field `LetterBody`.

```
openresource "Letter Templates"
LetterBody=getresource("TEXT","Thank You #2")
```

Reading STR and STR# Resources

Panorama has three special functions for reading with string resources: `getstring()`, `getnstring()` and `getstringmatch()`.

To read a `STR` resource (which contains one text item up to 255 characters long), use the `getstring()` function. This function has two parameters: `type` and `ID`. `Type` is the resource type. This must be a four letter text item (see “[Working with Resources](#)” on page 433). You can specify any resource type you like here, but strings are usually stored in resources of type `"STR "` (Pascal String). (If you specify `"` for the type, Panorama will assume `"STR "`.) The `ID` parameter is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).

This example displays the contents of STR resource number 1296.

```
message getstring("",1296)
```

All resource have numbers, but they do not all have names. If the resource does have a name, you can use the name for the ID. This example displays the text in the resource named `Overflow Error`.

```
message getstring("","Overflow Error")
```

To access a `STR#` resource(which contains multiple text items, each up to 255 characters), use the `getnstring()` function. `STR#` resources hold multiple text items, so the `getnstring()` function extracts one of them. This function has three parameters: `type`, `ID` and `number`. The `type` is the resource type. This must be a four letter text item. You can specify any resource type you like here, but strings are usually stored in resources of type `"STR#"` (multiple Pascal Strings). (If you specify `"` for the type, Panorama will assume `"STR#"`.) The `ID` is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item). The `number` is the number of the string item within the collection. For example, if the collection contains 6 strings they will be numbered 0, 1, 2, 3, 4, and 5.

This example displays the contents of STR# resource #693 item 12.

```
message getnstring("",1296,11)
```

Here is another example that displays the 12th item in the `Errors` STR# resource.

```
message getnstring("","Errors",11)
```

Another function that access **STR#** resources is the `getstringmatch()` function. The `getstringmatch()` function searches through a collection of multiple strings in a **STR#** resource. If it finds a match with the text you supply, it returns the number of the text item within the collection.

The `getstringmatch()` function has three parameters: `type`, `ID` and `text`. The `type` and `ID` are the same as for the `getnstring()` function (see above). The `text` parameter is the text you want to search for. For a match, this text must be exactly the same as one of the text items in the **STR#** collection.

This function returns a number. If the text does not match any of the text items in the **STR#** collection, the function will return 0. If there is a match, the function will return the number of the item that matched, starting with 1 for the first item. (Notice that this numbering system is different than the `getnstring()` function, which starts with 0 for the first item.)

One application for this function is looking up commands or keywords. Suppose you have a **STR#** resource number 320 that contains the following text items.

```
DIAL
APPOINTMENT
TODO
LETTER
CHECK
```

Now suppose the database has a global variable called `CommandLine`. The user types a command into this variable with a Text Editor SuperObject™. Here is part of a procedure that can process these commands using the **STR#** 320 resource.

```
local commandWord, commandNumber, commandExtras
openresource "Accounting Extras"
commandWord=upper(strip(CommandLine[1," "]))
commandExtras=strip(CommandLine["",-1])
commandNumber=getstringmatch(" ",320,commandWord)
if commandNumber=0 stop endif
if commandNumber=1
    dial commandExtras
endif
if commandNumber=2
    ...
endif
if commandNumber=3
    ...
    ...
```

Notice that the example converts the text the user types in into all upper case. This is to make sure that the text will match the commands in the **STR#** resource. Remember, the text must match exactly, including upper and lower case.

Writing a Resource

If a resource file has been opened with the `openresourcerw` statement a procedure can use the `writeresource` statement to create and/or modify a resource object. The `writeresource` statement has three parameters: `type`, `ID` and `data`. The `type` is the resource type. This must be a four letter text item (see “[Working with Resources](#)” on page 433). The `ID` is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item). The `data` parameter is the data to be placed in the resource. The `data` should be text, not a number. (The text may represent a binary value, see “[Raw Binary Data](#)” on page 156.)

Here is a procedure that writes some text (for example `Last update: 10/18/02`) into **STR** resource number 2000. If the resource does not exist it will be created.

```
writeresource "STR ",2000,string255("Last update: "+datepattern(today(),"mm/dd/yy") )
```

If there is more than one resource file open the resource will be written into the file that was most recently opened (see “[Working with Multiple Resource Files](#)” on page 440).

Deleting a Resource

If a resource file has been opened with the `openresourcerw` statement a procedure can use the `deleteresource` statement to permanently remove a resource .

```
deleteresource type,id
```

The `type` is the resource type. This must be a four letter text item (see “[Working with Resources](#)” on page 433). The `ID` is the identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item).

Renumbering a Resource

If a resource file has been opened with the `openresourcerw` statement a procedure can use the `renameresource` statement to change the number of a resource and/or change its name (see “[RENAMERESOURCE](#)” on page 5659).

```
renameresource type,id,number,name
```

The `type` is the resource type. This must be a four letter text item (see “[Working with Resources](#)” on page 433). The `ID` is the original identification for the resource. The resource id can be a number (from 0 to 65,535) or a name (a text item). The number is the new `number` for the resource. The `name` is the new name for the resource. This procedure changes resource `MENU 498` into `MENU 1917`, and gives the resource the name `Option Menu`.

```
renameresource "MENU",498,1917,"Option Menu"
```

If the name is "" Panorama will leave the name alone (no change). This procedure changes resource `MENU 498` into `MENU 8367` while leaving the name alone (as `Option Menu` in this case).

```
renameresource "MENU",1917,8367,""
```

If you want to change the name to empty text then use ¶ as the menu name (see “[Special Characters](#)” on page 57). This procedure erases the name from resource `MENU 8367`.

```
renameresource "MENU",8367,8367,¶
```

Listing Resources

Panorama has two functions that can help you build a list of the resources available in the currently open resource files: `resourcetypes()` and `resources()`.

The `resourcetypes()` function creates a text array containing a list of the resource types in the currently open resource files. The function has no parameters.

The `resourcetypes()` function returns a carriage return delimited text array (see “[Text Arrays](#)” on page 93). Each element in the array contains a resource type. Each resource type is a four letter text item, for example `"STR "` (Pascal String), `"STR#"` (multiple strings), `"DLOG"` (dialog), `"DITL"` (dialog items), `"MENU"` (menu) (see “[Working with Resources](#)” on page 433).

You can use this function to check if a particular resource type exists, or you can use the function with a pop-up menu or List SuperObject™ to allow the user to select a type of resource for any reason. The procedure below will create a text array with resource types.

```
local rezTypes
rezTypes=resourcetypes()
```

The `rezTypes` variable will be filled with a list of resource types, like this:

```
CNTL
CURS
FKEY
INIT
KCAP
KCHR
LDEF
MACA
PACK
PTCH
ROv#
TPLT
SIZE
LBAR
octb
DLGX
dctb
cocm
TEXT
STR#
PICT
PAT#
PAPR
MENU
MDEF
```

As you can see, the resource types are not listed in any particular order.

The `resources()` function creates a text array containing a list of resources of a particular type. This function has one parameter: `type`, which is the resource type. This must be a four letter text item (see “[Working with Resources](#)” on page 433).

The `resources()` function returns a text array containing a carriage return delimited list of all the resources of the specified type. Each element of this list is itself a tab delimited array. The first item is the resource item number. The second item is the resource name (if any).

This example builds a list of the `TEXT` resources in the currently open resource files. (The currently open resource files include Panorama itself and the Macintosh system file, as well as any resource files you have opened with the `openresource` statement.)

```
local rezStrings
rezStrings=resources("TEXT")
```

This will fill `rezStrings` with an array like this. (There is a tab between the number and the first character of the resource name, if any.)

```
2001Error Messages
2002Command List
2003Conversion Options
2100
2103
2104
2140Day
2141Month
2142Year
```

The `resourcetypes()` and `resources()` functions normally list all resources in all open resource files. See the next section to learn how to make these function list only the resources in a single file.

Working with Multiple Resource Files

It's possible to open and work with multiple resource files at once. For example you could open three resource files like this.

```
openresourcerw "alpha"
openresourcerw "beta"
openresourcerw "gamma"
```

When reading resources Panorama always searches the most recently opened file first. For example, if a procedure contained the statement

```
temp=getresource("DATA",2000)
```

Panorama would start by searching for this resource object in the **gamma** resource file. If it didn't find it there it would look in the **beta** resource file, and if not there then it would look in the **alpha** file.

When writing resources Panorama always writes in the most recently opened file. For example if a procedure contained the statement

```
writeresource "DATA",2000,"This is not a test"
```

Panorama would always write this resource in the **gamma** resource file.

Sometimes it may be necessary to focus in on only a single resource file, temporarily disabling the other open files. This can be done with the **activeresource** statement. This statement temporarily makes only one resource file active. For example, the program below will write a resource into the beta resource file instead of the gamma file.

```
activeresource "beta"
writeresource "DATA",2000,"This is not a test"
activeresource ""
```

The second **activeresource** statement (with the "" parameter) re-enables the other resource files.

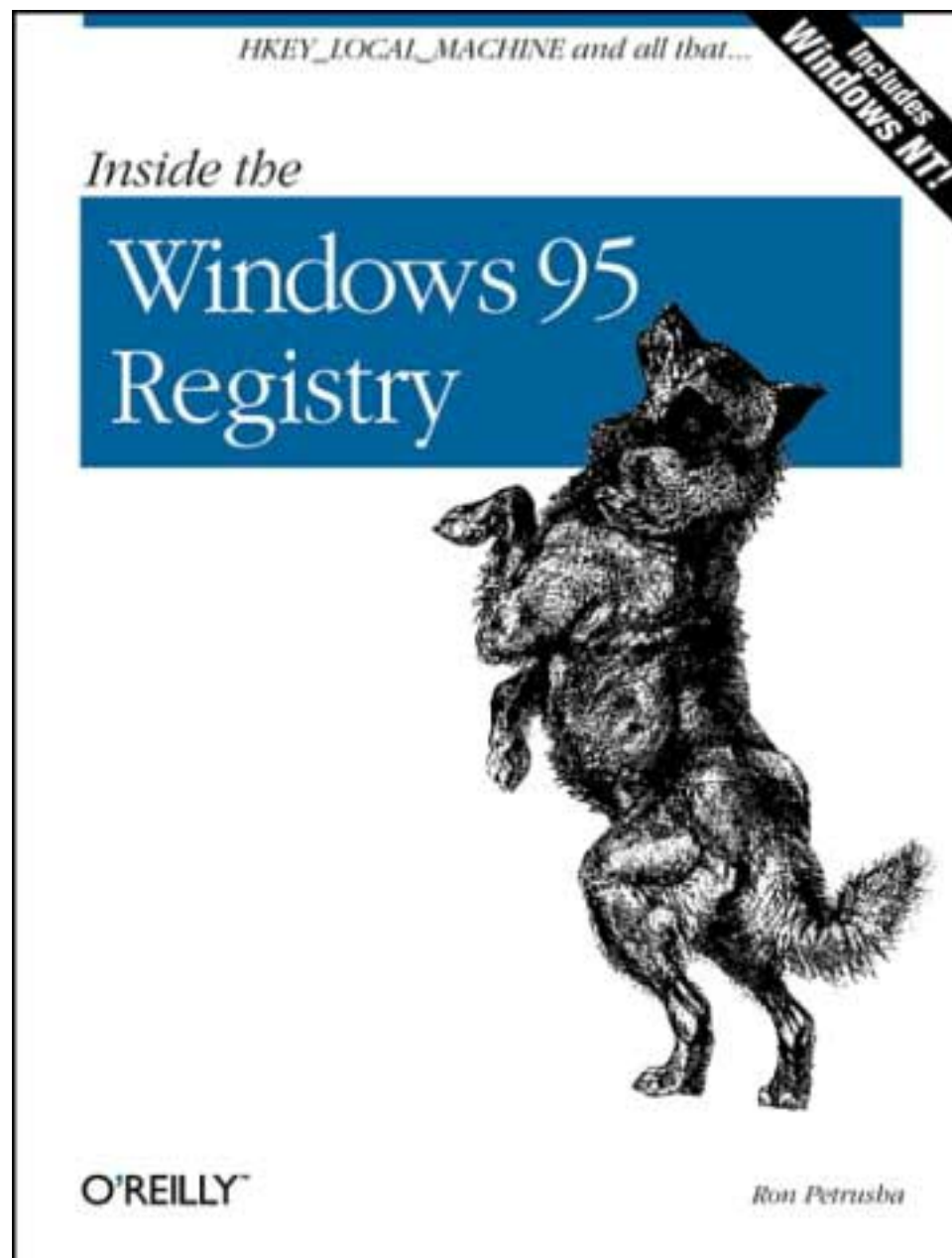
The **activeresource** statement can also be used for reading resources. The program below will only read the DATA 2000 resource from the **beta** file, not **alpha** or **gamma**.

```
activeresource "beta"
temp=getresource("DATA",2000)
activeresource ""
```

Finally, the **activeresource** statement can also be used when listing resources. The **resourcetypes()** and **resources()** functions normally list all resources in all open resource files (see "[Listing Resources](#)" on page 438). When an **activeresource** statement is in use they will only list resources and resource types in the active resource file, ignoring the other files.

Accessing the Windows Registry

The **Registry** is a master database that Windows uses as kind of a giant system-wide preferences file. If you'd like to learn about what's in the registry and how to work with it, I recommend that you pick up a copy of "Inside the Windows 95 Registry" by Ron Petrusba.



Be sure you know what you are doing before you mess with the Registry. You can easily disable your system beyond repair if you are not careful.

Getting Information About Registry Items

The `registryinfo()` function allows you get a directory of subkeys, a directory of values within a registry key, or a specific value within a key. For example, this formula returns a list of control panels.

```
registryinfo("HKEY_CURRENT_USER\Control Panel")
```

To get a directory of the values contained in a key (instead of the subkeys), add a colon to the end of the path. This example will list three values: `MouseSpeed`, `MouseThreshold1`, and `MouseThreshold2`.

```
registryinfo("HKEY_CURRENT_USER\Control Panel\Mouse:")
```

To retrieve a specific value, add the value name to the end of the path.

```
registryinfo("HKEY_CURRENT_USER\Control Panel\Mouse:MouseSpeed")
```

To get the default value for the key, use the name `<DEFAULT>`. This example will tell you that a `.aif` file is a **QuickTime** movie file (if you have QuickTime installed).

```
registryinfo("HKEY_CLASSES_ROOT\*.aif:<DEFAULT>")
```

Panorama allows the six root keys to be abbreviated, as shown in the table below.

Root	Abbreviation
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKUS
HKEY_CURRENT_CONFIG	HKCC
HKEY_DYN_DATA	HKDD

Using these abbreviations the examples given previously above could be rewritten as shown below:

```
registryinfo("HKCU\Control Panel")
registryinfo("HKCU\Control Panel\Mouse:")
registryinfo("HKCU\Control Panel\Mouse:MouseSpeed")
registryinfo("HKCR\*.aif:<DEFAULT>")
```

Modifying Registry Entries

The `registrywrite` statement allows you to create registry keys and registry values, or to modify an existing registry value. This statement has three parameters.

```
RegistryWrite path,type,data
```

The first parameter is a registry path. This path uses the same format as the `registryinfo()` function (see [“Getting Information About Registry Items”](#) on page 441).

The second parameter is the type of data being written. The possible choices are shown below. If you specify `""`, Panorama will default to `REG_SZ` (text).

0	REG_NONE
1	REG_SZ
2	REG_EXPAND_SZ
3	REG_BINARY
4	REG_DWORD
5	REG_DWORD_BIG_ENDIAN
6	REG_LINK
7	REG_MULTI_SZ
8	REG_RESOURCE_LIST
9	REG_RESOURCE_LIST
10	REG_RESOURCE_REQUIREMENTS_LIST

The third parameter is the actual data being written. No matter what data format you are writing, this should be text. For other data types you can fill the text item with binary values (see [“Raw Binary Data”](#) on page 156).

This example changes the mouse speed. Notice that this statement supports the same abbreviations allowed by the `registryinfo()` function (see “[Getting Information About Registry Items](#)” on page 441).

```
registrywrite "HKCU\Control Panel\Mouse:MouseSpeed", "", "2"
```

You can also change the default value associated with a registry key.

```
registrywrite "HKCR\*\*.aif:<DEFAULT>", "", "Quick Time Movie"
```

This example creates a registry entry named `Acme`, but does not create or modify any values associated with that key.

```
registrywrite "HKLM\Software\Acme", "", ""
```

Deleting a Registry Entry

The `registrydelete` statement may be used to delete a registry key or registry value. This statement has one parameter, the path of the registry item to be deleted (see “[Getting Information About Registry Items](#)” on page 441).

This example deletes a registry value:

```
registrydelete "HKLM\Software\Acme\SuperWidget:WindowLocation"
```

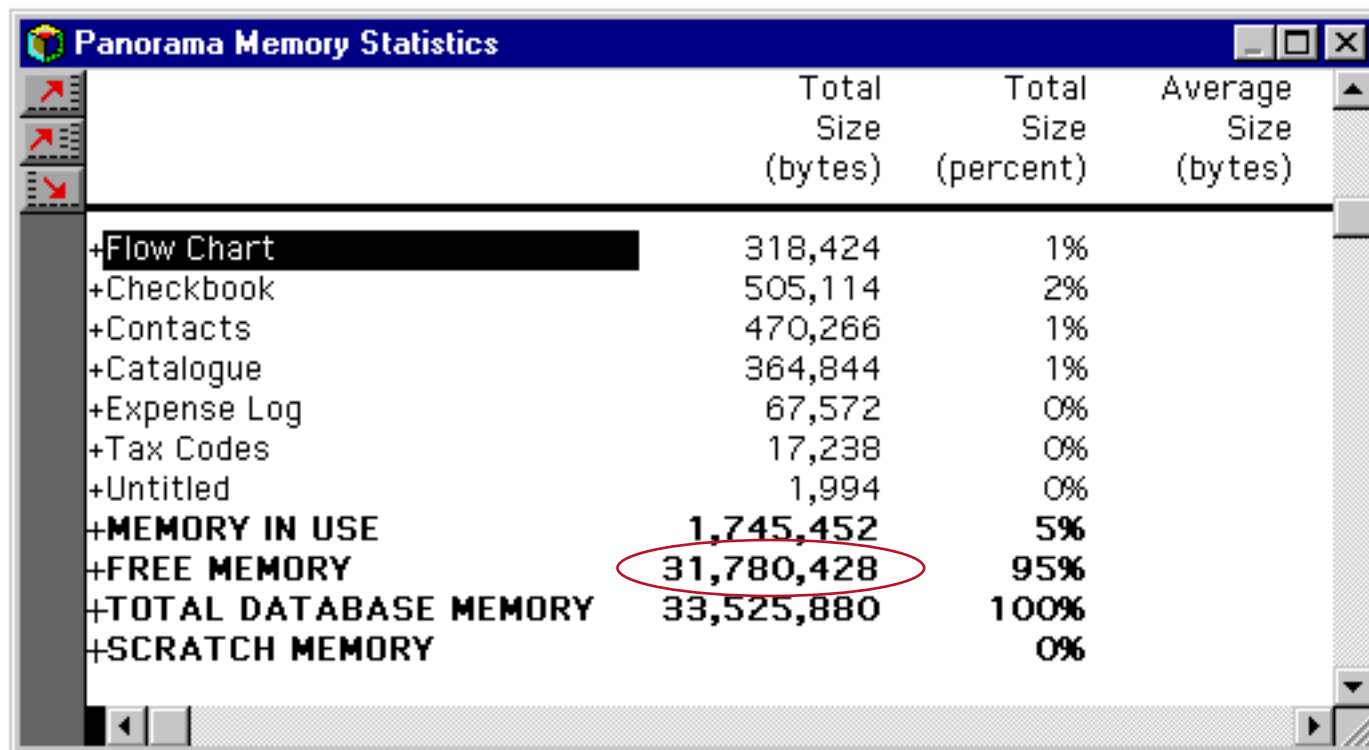
This example deletes a registry key, along with any values associated with it:

```
registrydelete "HKLM\Software\Acme\SuperWidget"
```

Monitoring Memory Usage

Since Panorama is a memory based program memory usage can be very important. Several statements allow you to monitor memory usage.

The `info("freememory")` function returns the amount of memory available for additional data. This corresponds to the value shown by the Memory Usage window (see "[Monitoring Memory Usage](#)" on page 137 of the *Panorama Handbook*).



	Total Size (bytes)	Total Size (percent)	Average Size (bytes)
+Flow Chart	318,424	1%	
+Checkbook	505,114	2%	
+Contacts	470,266	1%	
+Catalogue	364,844	1%	
+Expense Log	67,572	0%	
+Tax Codes	17,238	0%	
+Untitled	1,994	0%	
+MEMORY IN USE	1,745,452	5%	
+FREE MEMORY	31,780,428	95%	
+TOTAL DATABASE MEMORY	33,525,880	100%	
+SCRATCH MEMORY		0%	

To learn how to change Panorama's overall memory allocation see "[Adjusting Panorama's Memory Allocation](#)" on page 140 of the *Panorama Handbook*.

The `info("scratchmemory")` function returns the amount of scratch memory available. On Macintosh OS X and Windows PC systems there is no scratch memory (this is a relic of OS 9), so this function always returns 1,000,000 (one million).

Windows

Windows are where the action is. Except for the menu bar, everything Panorama does happens inside of windows. A procedure can open windows, close them, move them around and control their appearance.

Opening a Window

Opening a window in a procedure generally requires at least two statements. The first statement sets up the location of the window on the screen, along with any window options. The second statement actually opens the window.

There are six primary statements that open a new window: `opensheet`, `openform`, `opendialog`, `opencrosstab`, `openprocedure` and `opendesignsheet`. The procedure can specify exactly where the new window will open (see the next section) or it can simply allow Panorama to open the window using the default location and size.

The `opensheet` statement opens the data sheet for the current database in a new window. If the data sheet window is already open, that window will simply be brought to the front (no new window will be opened).

The `openform` statement opens a form in a new window. The specified form must be a form in the current database, not some other database. The `openform` statement has one parameter, the name of the form to open. The procedure below opens a form in a new window, prints a report, then closes the new window.

```
openform "Monthly Report"
field PayTo
groupup
field Amount
total
print dialog
removesummaries 7
closewindow
```

If the specified form is already open in a window, that window will simply be brought to the front (no new window will be opened).

A procedure can use the `dbinfo` function to find out what forms (or crosstabs or procedures) are in a database. The procedure below opens up every form in the current database.

```
local windowCount,formCount,nextForm
windowCount=arraysize(listwindows(""),¶)
formCount=arraysize(dbinfo("forms",""),¶)
if windowCount+formCount>23
    message "Too many forms, cannot open them all"
    formCount=23-windowCount
endif
nextForm=1
loop
    openform array(dbinfo("forms",""),nextForm,¶)
    nextForm=nextForm+1
until nextForm>formCount
```

The `opendialog` statement also opens a form in a new window. However, this statement opens the window without any drag bar, tool palette, or scroll bars. In other words, the new window will look (and act) like an old style, non-movable dialog window. Since this type of dialog is now obsolete we recommend that you avoid the `opendialog` statement. When a form is opened with the `opendialog` statement, Panorama will not allow any other window to be moved on top of the dialog window. Panorama simply ignores clicks in other windows, as well as clicks on the desktop. The only way to close this window is with the

closewindow statement. The form should have at least one button that triggers a procedure that will close the window. The example below opens a dialog window, then pauses. In this case the window should contain at least one button that has a **resume pState** statement in it, and it should have SuperObjects™ for editing the **PrintStartDate** and **PrintEndDate** global variables.

```
global pState
setwindowrectangle rectangle(100,150,300,450)," "
opendialog "Print Options"
pause pState
closewindow
select Date>=PrintStartDate and Date<=PrintEndDate
print dialog
```

For information on a better way to create dialogs with Panorama forms see “[Custom Dialogs](#)” on page 489.

The **opencrosstab** statement opens a crosstab in a new window. The specified crosstab must be a crosstab in the current database, not some other database. The **opencrosstab** statement has one parameter, the name of the crosstab to open.

The **openprocedure** statement opens a procedure in a new window, so that the procedure can be edited. The procedure must be a procedure in the current database, not some other database. The **openprocedure** statement has one parameter, the name of the procedure to open.

The **opendesignsheet** statement opens the design sheet for the current database in a new window. If the design sheet window is already open, that window will simply be brought to the front (no new window will be opened). Once the design sheet is open the procedure can modify the structure of the database (be careful!). A procedure can also modify the structure of the database with the **fieldname**, **fieldtype**, **addfield**, **insertfield** and **deletefield** statements (see “[Modifying Field Structure Directly](#)” on page 506).

Specifying the New Window Location

There are three statements that can define the location and options for a new window: **setwindowrectangle**, **setwindow** and **windowbox**. The **setwindowrectangle** statement has two parameters: a **rectangle** defining the location of the new window (relative to the upper left hand corner of the main screen, see “[Rectangles](#)” on page 149), and the window **options**. (The window options will be discussed in the next section, but if you use " " you will get a standard window.)

Here is a procedure that opens a 100 by 200 pixel form window in the upper left hand corner of the main screen.

```
setwindowrectangle rectangle(25,2,125,202)," "
openform "Status"
```

If you want the new window to fill the entire screen you can use the **info("screenrectangle")** function, like this:

```
setwindowrectangle info("screenrectangle")," "
openform "Status"
```

The **setwindowrectangle** statement will allow you to open the window partially or completely off the screen, where it won't be visible. If you don't want that to happen you can use the **fitwindow** statement in between the **setwindowrectangle** statement and the statement that opens the new window (**openform**, **opensheet**, etc.). The **fitwindow** statement adjusts the new window position to make sure that it is com-

pletely visible on the screen. The procedure below positions the **Status** window in the lower right hand corner of an XGA (1024 by 768) monitor. But on an older 640 by 480 monitor the window would be partially off the screen. The **fitwindow** statement adjusts the window location so it will be completely visible even on these small monitors.

```
setwindowrectangle rectangle(568,624,768,1024), ""
fitwindow
openform "Status"
```

You'll often want to center a new window on top of the current, window, or center a new window on the screen. The subroutine procedure below, called **.CenterRectangle**, is handy for these situations.

```
local newRect,oldHeight,oldWidth,newHeight,newWidth
local newTop,newLeft
oldHeight=rheight(parameter(1))
oldWidth=rwidth(parameter(1))
newHeight=parameter(2)
newWidth=parameter(3)
newTop=max(20,rtop(paramter(1)+int((oldHeight-NewHeight)/2))
newLeft=rleft(parameter(1)+int((oldWidth-newWidth)/2)
newRect=rectanglesize(newTop,newLeft,newHeight,newWidth)
setparameter 1,newRect
```

Once you've added the **.CenterRectangle** procedure to your database you can use it to open windows. Here's a procedure that centers a new 200 pixel high by 450 pixel wide window in the middle of the screen.

```
local myWindowBox
myWindowBox=info("screenrectangle")
call .CenterRectangle,myWindowBox,200,450
setwindowrectangle myWindowBox, ""
openform "Schedule"
```

This next procedure will center the new window in the middle of the current window.

```
local myWindowBox
myWindowBox=info("windowrectangle")
call .CenterRectangle,myWindowBox,200,200
setwindowrectangle myWindowBox, ""
openform "Options"
```

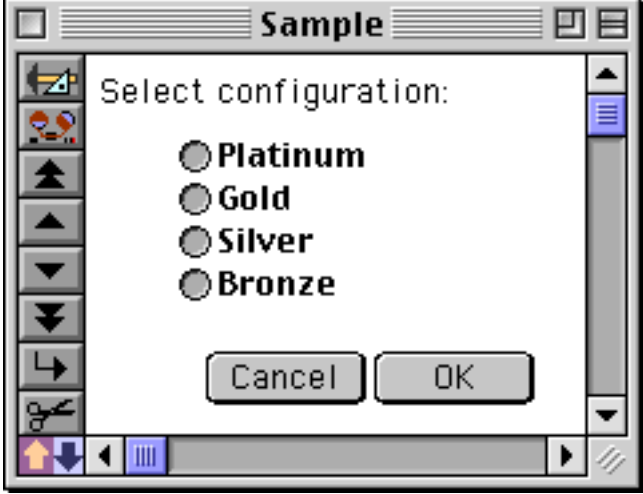
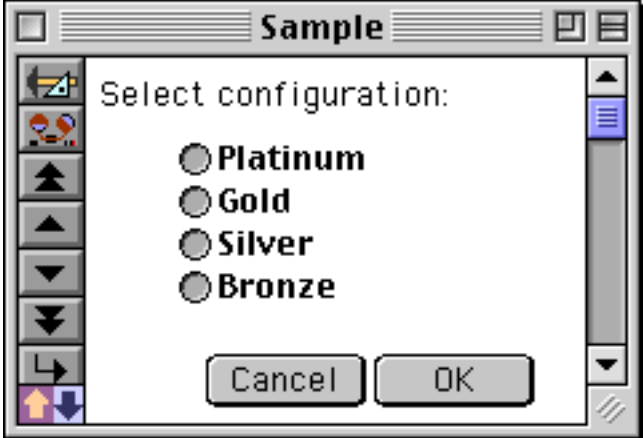
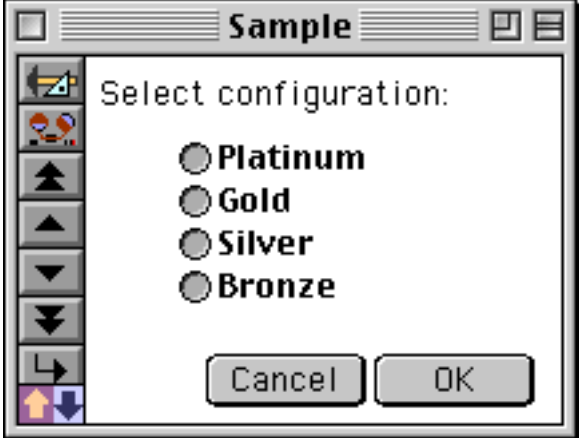
The **setwindow** statement is similar to **setwindowrectangle**, but uses four separate numbers as parameters instead of using a rectangle. The **windowbox** statement also uses four numbers, but they are combined into a single text item. (Also, the **windowbox** statement only has one parameter, so you cannot set the window options.) The following three statements will work identically to each other.

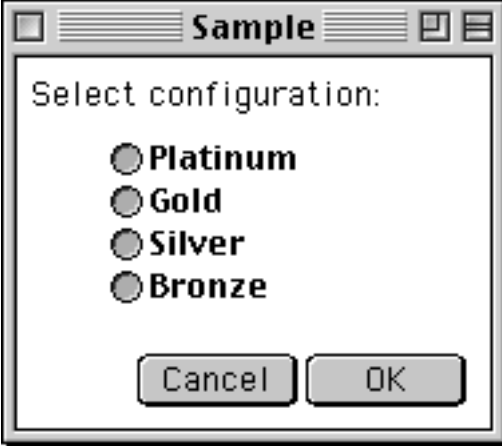

```
setwindowrectangle rectangle(100,120,250,400), ""
setwindow 100,120,250,400, ""
windowbox "100 120 250 400"
```

You may wonder why Panorama has three statements for doing the same thing. The answer is that as newer statements are added to Panorama in new versions, the older statements are retained for compatibility with existing procedures. In general, the **setwindowrectangle** statement is the best for new procedures because of all the functions Panorama now has for manipulating rectangles as a single unit (instead of having to deal with four separate numbers). See "[Rectangles](#)" on page 149 to learn more about these functions.

New Window Options

Normal Panorama windows have a drag bar across the top, a tool palette down the left hand side, and two scroll bars down the right and bottom edges of the windows. Using the `setwindowrectangle` or `setwindow` statement, a procedure can open a window with or without any or all of these elements. The second parameter of these statements is a text argument that allows you to suppress these four window elements. The text may include any combination of these four components `nodragbar`, `nopalette`, `novertscroll` and `nohorzscroll`. Here are some examples of combinations of these options.

Option	Example
<p style="text-align: center;">""</p>	
<p style="text-align: center;">"nohorzscroll"</p>	
<p style="text-align: center;">"nohorzscroll novertscroll"</p>	

Option	Example
<pre>"nopalette nohorzscroll novertscroll"</pre>	
<pre>"nodragbar nopalette nohorzscroll novertscroll"</pre> <p style="color: red; text-align: center;">NON-MOVABLE WINDOWS ARE NO LONGER RECOMMENDED!</p>	

Here is a procedure that opens a 100 by 200 pixel form window in the upper left hand corner of the main screen. The window has no tool palette and no scroll bars.

```
setwindowrectangle rectangle(25,2,125,202),"nopalette novertscroll nohorzscroll"  
openform "Status"
```

You can use these options to open a form as a dialog window with no drag bar, no tools, and no scroll bar. Another way to do this is with the `opendialog` statement, which is described later in this chapter. If a window is opened with the `opendialog` statement Panorama will treat it as a dialog and will not allow any other window to be brought on top of it (including windows of other currently running applications).

Non Standard Window Styles

In addition to standard windows Panorama supports several custom window styles. In fact, if you are using a Macintosh computer and you are an advanced C, Pascal or assembly language programmer you can even create your own window types. The `windowproc` statement allows you to override the normal default window type and use any type of window.

The `windowproc` statement should be placed just in front of the statement that actually opens the window (see the example below). The `windowproc` statement has one parameter—a window type number.

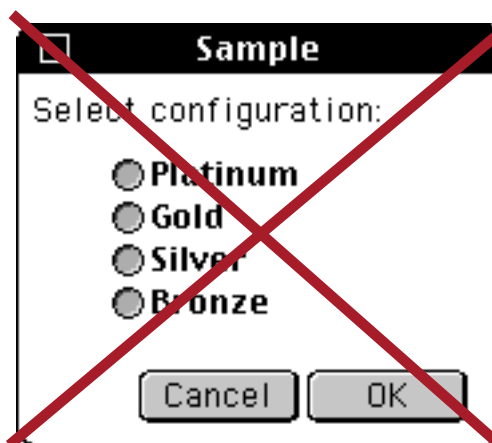
Window type 2 is a plain dialog box with no border and no drop shadow.



Window type 3 is also a plain dialog with no border. However, type 3 windows do have a drop shadow.



In previous versions of Panorama window type 16 was a rounded corner window with a black drag bar. This window type is no longer available.



Once a window has been opened, the window type is permanent. The only way to change a window type is to close the window and then re-open it with a new type.

Changing a Window's Position/Options

If a window has a drag bar, the user can move the window around on the screen. If you just want to change the size of a window without moving it use the `zoomwindow` statement. This statement has two parameters:

```
zoomwindow height,width
```

Using the `zoomwindow` statement a procedure can also move a window, and/or change the display options for that window. The `zoomwindow` statement has five parameters:

```
zoomwindow top,left,bottom,right,options
```

These are the exact same options used by the `setwindow` statement.

The procedure below moves the current window to the right by 50 pixels. (The example assumes the current window has standard window options.)

```
local deltaV,deltaH
deltaV=0
deltaH=50
zoomwindow
  deltaV+rtop(info("windowrectangle")),
  deltaH+rleft(info("windowrectangle")),
  deltaV+rbottom(info("windowrectangle")),
  deltaH+rright(info("windowrectangle")),
  "" /* change this line to use non-standard window options */
```

If the window was originally close to the right hand edge of the screen, this procedure may push the window partially off the screen. If you want to prevent this, place a `fitwindow` statement just in front of the `zoomwindow` statement.

Changing a Window's View

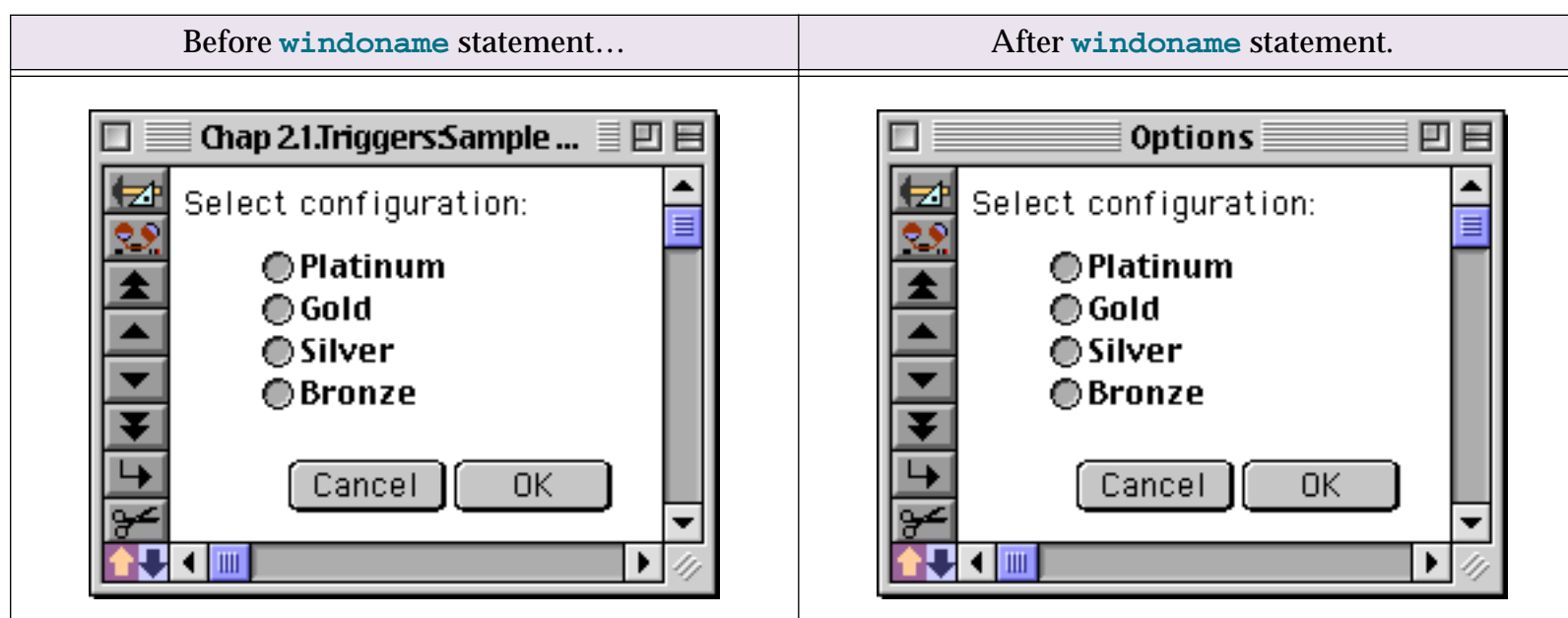
A procedure can change what is inside a window at any time. There are five primary statements that change what appears in the current window: `gosheet`, `goform`, `gocrosstab`, `goprocedure` and `godesignsheet`. These statements are the same as `opensheet`, `openform`, etc. except that instead of opening a new window, they simply open the requested form, crosstab, etc. inside the current window. If the requested view is already open in another window (or even the current window) the go commands simply bring that window to the front. In that case, the original window will continue to display whatever it was displaying before.

The procedure below switches the current window to a form named `List`.

```
goform "List"
```

Changing the Name of a Window

Panorama windows usually have a name that combines the database name with the name of the form, crosstab, or procedure being displayed. However, using the `windowname` statement a procedure can rename any window to anything you want.



The `windowname` statement has one parameter, the name for the window. The only restriction is that the name must be less than 31 characters long.

Suppose you have a database named `Team` with a form named `Status`. Normally the name of this window would be `Team:Status`. The procedure below opens the `Status` window and renames it to `Status Display`.

```
openform "Status"  
windowname "Status Display"
```

The new window name is only temporary. Panorama will forget the new window name if the window is closed, or if the contents of the window are changed to a different view (either with the `View` menu or with a `goform`, `gosheet`, or other `go<view>` statement.)

Scrolling Inside a Form Window

If a form is larger than the window, the user can scroll to different parts of the form using the scroll bars. Using the `formxy` statement, a procedure can also scroll the form within the window. This statement has two parameters, the vertical and horizontal position:

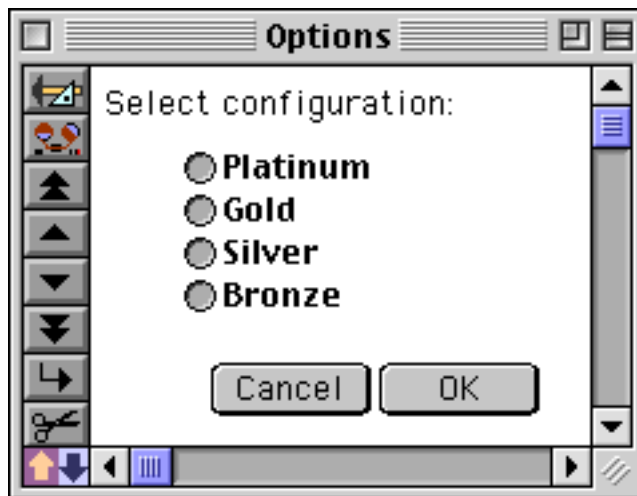
```
formxy vertical, horizontal
```

The `vertical` parameter is the vertical spot on the form that you want to appear at the very top of the window. If you want to see the top of the form this should be zero. The position is specified in pixels (1 pixel = 1/72 inch).

The **horizontal** parameter is the horizontal spot on the form that you want to appear at the very left edge of the window. If you want to see the left edge of the form, this should be zero.

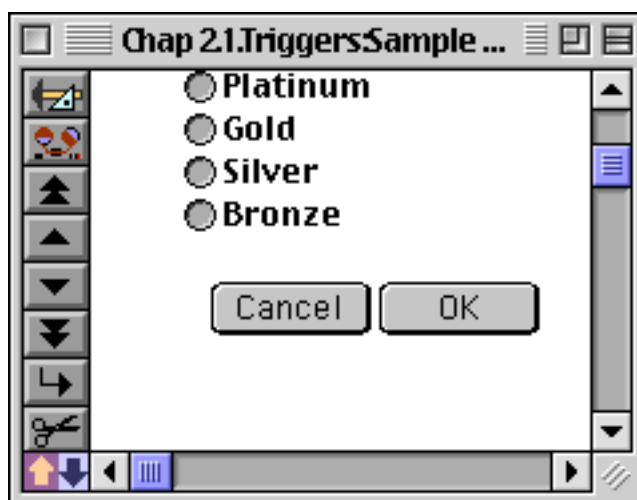
This simple example procedure makes sure that the upper left hand corner of the form is visible.

```
formxy 0,0
```



The next procedure slides the form down 1/2 inch.

```
formxy 72/2,0
```



Sometimes the **formxy** statement may not seem to slide the form to the exact position you specified. There are two possible reasons for this. First of all, the form cannot be scrolled farther than the bottom right object on the form. Once this object is visible, the form cannot be scrolled any farther.

The other possible discrepancy is that forms are always scrolled in increments of 8 pixels. This insures that patterns (which repeat every 8 pixels) are always displayed consistently. The **formxy** statement will always round the position you specify to the nearest multiple of 8.

Closing a Window

The user can usually close a window by clicking on the close box (in the upper left hand corner of the drag bar). A procedure can close a window with the **closewindow** statement. This statement has no parameters—it simply closes the current (top) window.

When Panorama closes the last window associated with a database, it normally asks the user if he or she wants to save changes before closing the database. However, the `closewindow` statement does not usually do this. It simply closes the window and the database without saving any changes. Sometimes this is very convenient. A procedure can open a database, sort it, select, etc., then print a report and close the database, throwing away the temporary changes it has made. However, if you want the procedure to ask the user if they want to save changes, there are two ways to do this. The first is to write this into the procedure itself.

```
if info("changes")>0 and arraysize(listwindows(info("databasename")))=1
  alert 1013,"Do you want to save changes?"
  if clipboard() contains "yes"
    save
  endif
endif
closewindow
...
... procedure continues
```

The second method is simply to make sure that the `closewindow` statement is either the last statement in the procedure, or that it is immediately followed by a `stop` statement. In either case Panorama will check for changes and ask the user if they want to save changes if necessary. (If `closewindow` is the last statement and you don't want Panorama to ask to save changes, simply put an extra `nop` statement (no-operation) after the `closewindow` statement.)

Trapping the Close Box

If the window has a drag bar, the user can close the window at any time simply by clicking on the close box. You may want to prevent the user from closing the window under certain conditions, or you may want to perform some special operation before the window is closed. These situations call for a `.CloseWindow` procedure (see "`.CloseWindow`" on page 380). Whenever the user clicks on a close box, Panorama checks to see if there is a `.CloseWindow` procedure in this database. If there is, Panorama triggers the procedure instead of closing the window.

For example, suppose you want to allow the user to close any window except for the data sheet window. This `.CloseWindow` procedure will do the trick:

```
if info("windowname")=info("databasename")
  message "Sorry, you cannot close the data sheet"
else
  closewindow
endif
```

Notice that if it wishes to close the window, the `.CloseWindow` procedure must explicitly use the `closewindow` statement.

It's important to keep it mind that the only time the `.CloseWindow` procedure is triggered is when the user clicks on the close box. It is not triggered when the user selects the `Quit` or `Close File` commands from the File menu, or by the `closewindow` statement.

In addition to trapping the close box, you may wish to trap the action of closing the entire database. You can do this with the `..CloseDatabase` procedure, see "`..CloseDatabase`" on page 397.

Changing The Window Order (Who's on Top?)

The user can bring any window to the front by clicking on the window. A procedure can bring a window to the front with the `window` statement. This statement has one parameter, the name of the window that needs to be brought to the front. The name must be spelled and capitalized exactly as it appears in the window title at the top of the window.

```
window "Price List:Report"
```

If there is no such window, the procedure will normally stop and display an error. The procedure can trap and respond to this error with the `if error` statement, like this (see “[Error Handling with if error](#)” on page 258).

```

window "Price List:Report"
if error
  openfile "Price List"
  openform "Report"
endif

```

Use the `info("windowname")` function to get the name of the currently active (top) window. One application for this function is when you want to jump to a different window, then jump back to the original window. This example jumps to the price list window, sorts the price list, then jumps back to the original window.

```

local wasWindow
wasWindow=info("windowname")
window "Price List:Prices"
field Description
sortup
window wasWindow

```

Use the `info("windows")` function to get a text array listing of open windows. The text array is carriage return separated (see “[Text Arrays](#)” on page 93), and it lists the windows in order from front to back (the top window is array element 1, the next window is element 2, etc.) The procedure below swaps the order of the top and second from top window.

```

window array(info("windows"),2,1)

```

Here is another procedure that brings the bottom window to the front. Using this procedure over and over again will cycle through all of the windows.

```

local windowCount
windowCount=arraysize(info("windows"),1)
window array(info("windows"),windowCount,1)

```

The `listwindows()` function is similar to `info("windows")` but allows you to list only the windows that belong to a particular database. This function has one parameter, the name of the database. (If you leave the database name as an "" empty string, the `listwindows()` function will list all windows, just like the `info("windows")` function.) The procedure below displays the number of open **Price List** windows.

```

local windowCount
windowCount=arraysize(listwindows("Price List"),1)
message "The Price List has "+str(windowCount)+" windows open."

```

The `windowtoback` statement is the opposite of the `window` statement. It sends a specified window all the way to the bottom of the pile.

```

window "Price List:Report"
print dialog
windowtoback "Price List:Report"

```

When the procedure is done with the **Price List:Report** window, it moves it out of the way, below all of the other windows. (Of course another option would be to close the window.)

Temporary “Invisible” Windows

Often a procedure needs to flip back and forth between windows in different databases. Although this gets the job done, it is very annoying and slow if the windows overlap each other.

To get around this problem a procedure can work with **secret windows**. From a procedure's point of view, a secret window is just like any other window that contains a form. The good news is that secret windows are invisible! A procedure can flip to a secret window in another database, perform some operation on that database (search, sort, etc.), then go back to the original window—all without the flashing and updating that usually occurs when you flip from window to window.

Secret windows are temporary. A secret window can be created by the **window** statement. The secret window ceases to exist as soon as the procedure brings another window to the top (or as soon as the procedure stops).

To create a secret window and make it the current window, use the **window** statement with the parameter **<database>:secret**. For example, to open a secret window for the **Price List** database use this statement:

```
window "Price List:Secret"
```

The **Price List** database must be open or this statement will not work. The word **secret** may be capitalized any way you want: **secret**, **Secret**, or even **SECRET**.

Here is a procedure that opens the price list in a secret window, sorts the price list, then goes back to the original window (all without any flashing).

```
local wasWindow
wasWindow=info("windowname")
window "Price List:Secret"
field PartDescription
sortup
window wasWindow
```

(Note: If your database actually contains a form named **Secret** (or **secret** or **SECRET**) and it is open, the window statement will bring this window to the front instead of creating a secret invisible window.)

Databases Without Windows

Most Panorama databases have at least one window at all times. However, it is possible to create a database that has no windows at all. Such a database can be used for lookups, or it can be used with secret windows (see "[Temporary "Invisible" Windows](#)" on page 454).

To create a database that has no windows, use the **Save As** command and check the **No Windows** option (see "[Saving Window Positions](#)" on page 64 of the *Panorama Handbook*). The next time you open this database (either by double clicking on it or with the **Open File** dialog) it will open without any windows.

To temporarily open a database without its normal windows use the **opensecret** statement. This statement is identical to the **openfile** statement (see "[Opening a Panorama Database](#)" on page 405), but it does not open the windows. (It also does not trigger the **.Initialize** procedure, see "[.Initialize](#)" on page 382.) The example below opens the **Price List** database without any windows.

```
opensecret "Price List"
```

If you open a database with **opensecret** and then later decide that you want the windows after all you can use the **openfile** statement to open the windows. This will also trigger the files **.Initialize** procedure if it has one (see "[.Initialize](#)" on page 382).

```
opensecret "Price List" /* open file without windows */
...
...
...
openfile "Price List" /* file is already open, just open the windows */
```

The `makesecret` statement makes all the windows for the current database vanish. The database is still open and can be used for lookups or with secret windows. (Note: Since all the windows for the current database vanish, some other database will be the top window after this statement.) Here is an example `.CloseWindow` procedure that makes the database invisible if the user closes the last window:

```
if arraysize(listwindows(info("databasename")),1)=1
  makesecret
else
  closewindow
endif
```

If this is not the last window for this database, the procedure simply closes the window. If it is the last window, it closes the window with `makesecret`, so the database is still open in memory.

To re-open a window in a database that has no visible windows you need to use a secret window. The procedure below opens the `Distributor` window in the `Price List` database.

```
window "Price List:Secret"
setwindowrectangle rectangle(20,20,300,180)," "
openform "Distributor"
```

If you want to actually close a file with no windows you must again use a secret window. This procedure removes the `Price List` file from memory.

```
window "Price List:Secret"
closefile
```

Using secret windows, a procedure can access an invisible database just as easily as a visible database. The procedure can read, modify, sort, or even save the database. There's no need to open a visible window unless the user needs to see the database.

“Magic” Windows

Panorama has a number of `info()` functions and graphic statements that work with the currently active window. In some circumstances you may want to use one of these functions or statements to work with one of the other open windows. Panorama's “magic window” feature allows you to temporarily designate any open window as the currently active window for use with these `info()` functions (see “[Window, Form and Report Information](#)” on page 188) and graphic statements (see “[Programming Graphic Objects on the Fly](#)” on page 633). The designated window doesn't actually move to the front, so you can use this feature without unnecessary window “flashing.”

There are two statements available for designating an open window as the “magic” active window.

```
magicwindow windowname

magicformwindow database,form
```

In either case the window or form must already be open. From this point on in the program all `info()` functions and graphic commands will refer to this window, instead of the “real” current window. To remove the magic window designation and go back to the “real” current window use the statement

```
magicwindow " "
```

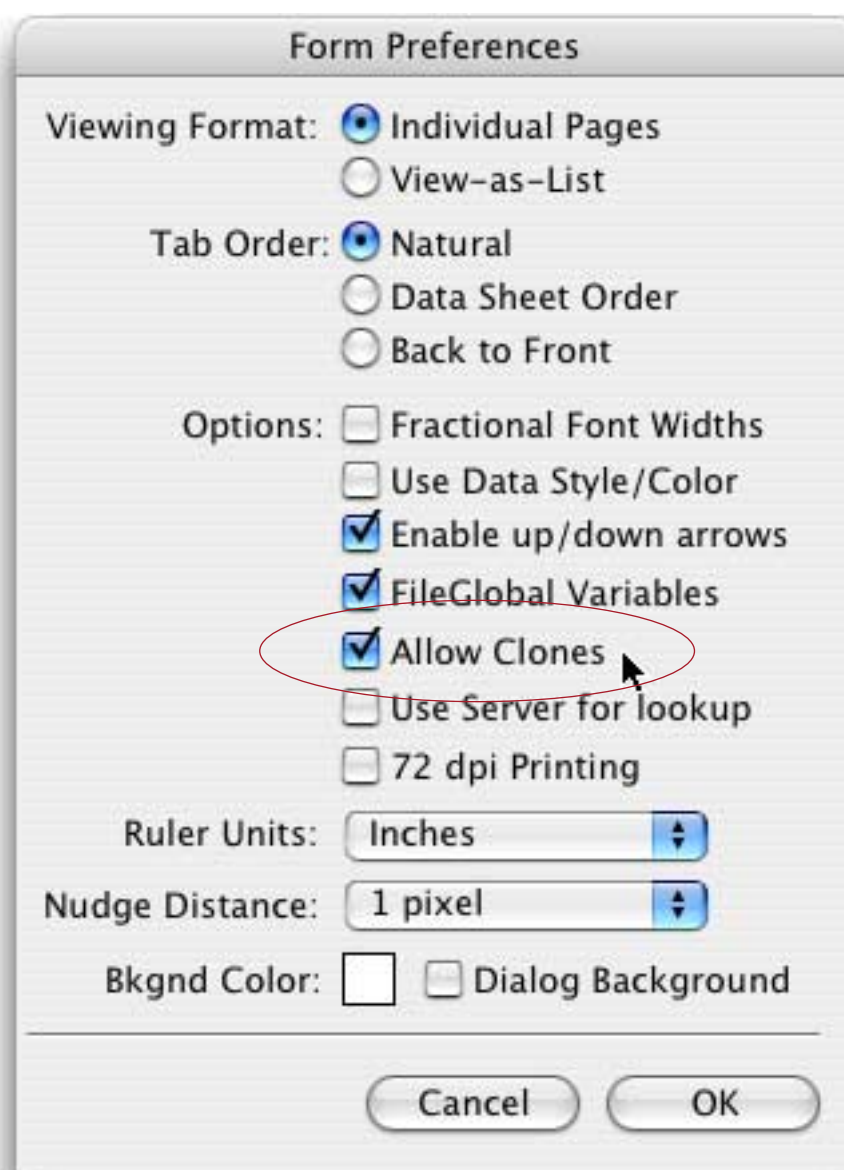

The `info("magicwindow")` function can be used to check the name of the currently designated “magic” window, if any. For example this little program will update the `People List` list object in the window `Contacts:List`. The `if` statement is used in case the specified window is not open at all.

```
magicwindow "Contacts:List"  
if info("magicwindow")<>" "  
    superobject "People List","Fill List"  
    magicwindow ""  
endif
```

You may confuse this “magic window” feature with the “secret window” feature, but they are quite different. The “secret window” feature creates a virtual, invisible window that is not tied to any actual window or form. This secret window may be used for any database operation, including data entry, searching, sorting, etc. The “magic window” feature does not create an invisible window, but works only with windows that are actually open. It does not affect most database operations (these still take place in the “real” active window), but only affects `info()` functions (see “[Window, Form and Report Information](#)” on page 188) and graphic statements (see “[Programming Graphic Objects on the Fly](#)” on page 633).

Window Clones

Panorama normally allows only a single window per form. However a form can be designed to be opened over and over again into multiple windows. This is called **window cloning**. To allow a form to be cloned you must open the **Form Preferences** dialog and select the **Allow Clones** option.



A window clone cannot be opened manually...clone windows must be created with the `openform` statement in a procedure. Here is a typical procedure that opens a slightly offset clone of the current window:

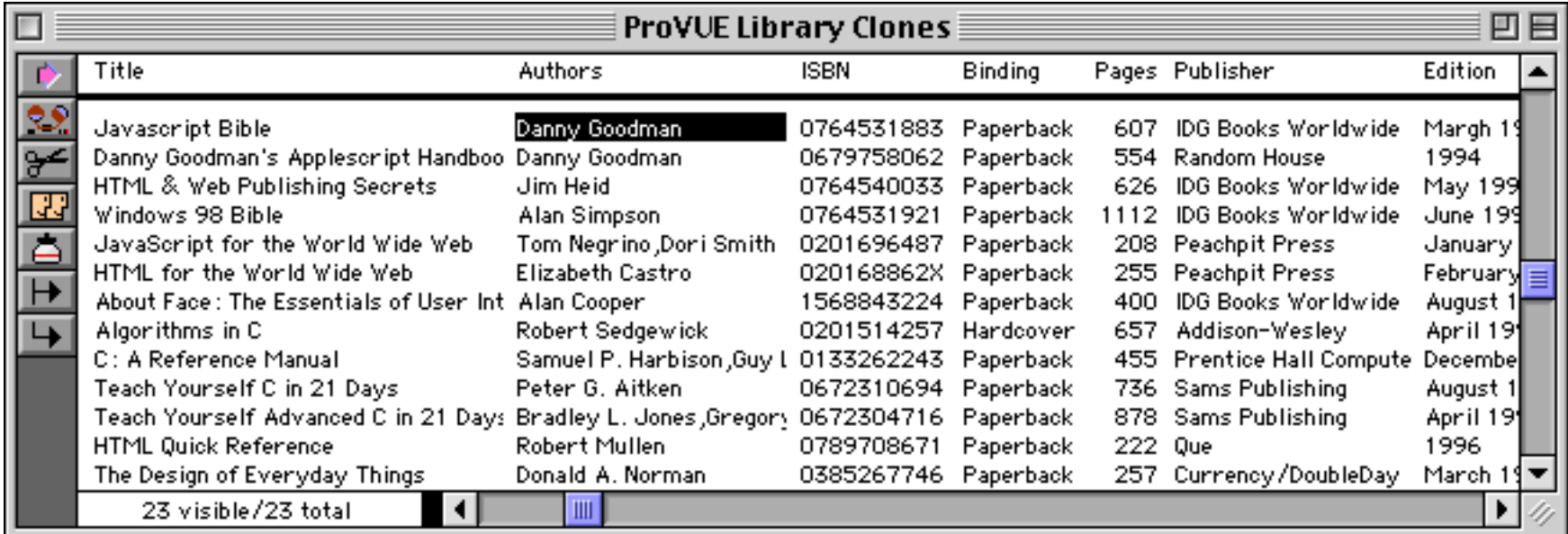
```
setwindowrectangle rectangleadjust(info("rectangle",10,10,10,10))
openform info("formname")
```

This procedure will not create a clone window unless the **Allow Clones** option is turned on.

Designing A Clone Window Application

Although any form can be cloned if the **Allow Clones** option is turned on, most forms will not work very intelligently if they are cloned. In general, a form that is designed to be cloned should not contain any fields or global variables, only `windowglobal` variables (see “[Variable Accessibility](#)” on page 250). If your form contains Text Editor, Data Button, Pop-Up Menu or List SuperObjects and the **Allow Clones** option is turned on, these SuperObjects will automatically create `windowglobal` variables instead of global variables. Since the `windowglobal` variables can be manipulated separately for each clone window you can control each clone window individually, even though all the clone windows use the same form template.

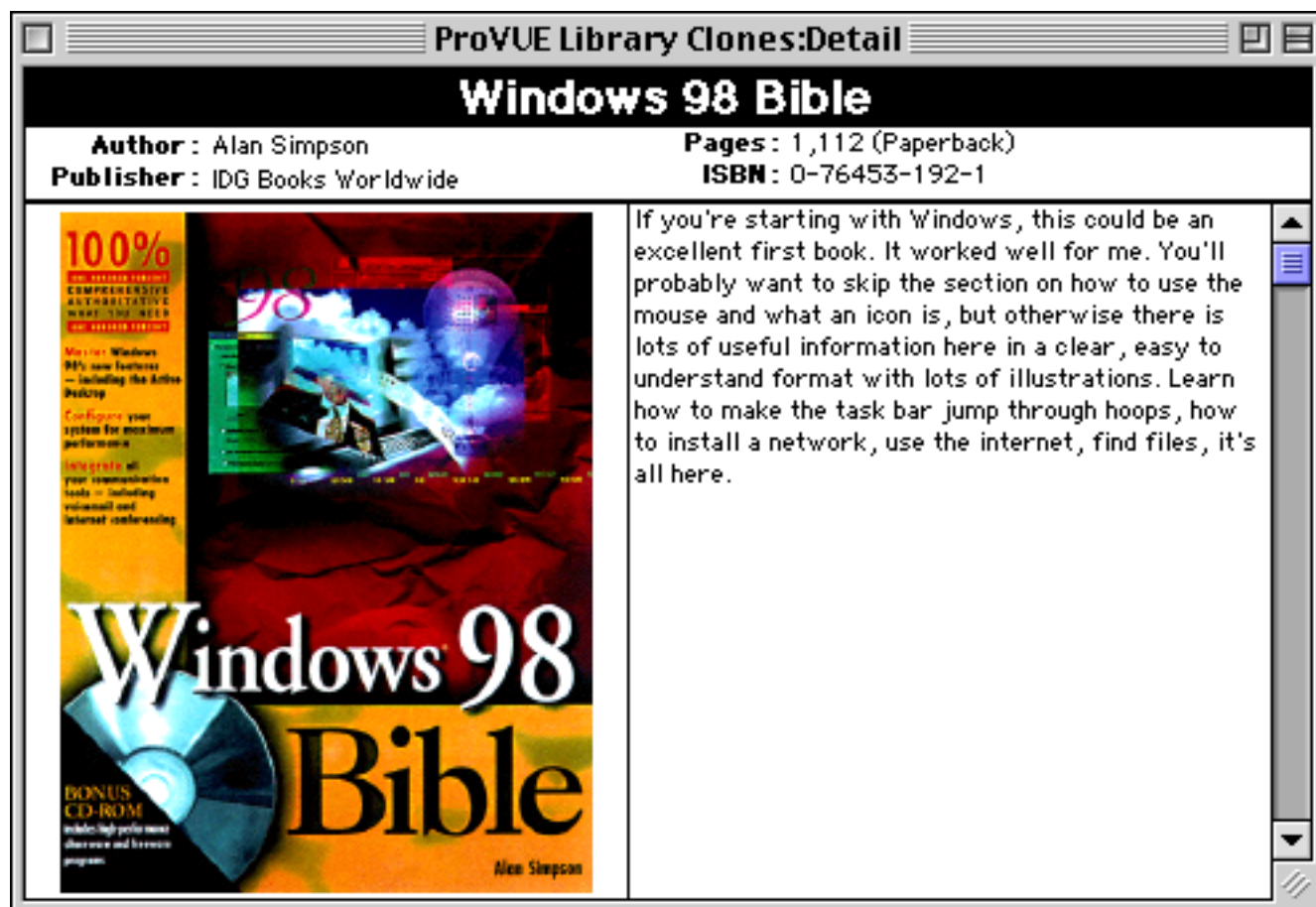
To illustrate clone windows we'll use this database that contains a list of books. Here is the data sheet.



Title	Authors	ISBN	Binding	Pages	Publisher	Edition
Javascript Bible	Danny Goodman	0764531883	Paperback	607	IDG Books Worldwide	March 1994
Danny Goodman's Applescript Handbo	Danny Goodman	0679758062	Paperback	554	Random House	1994
HTML & Web Publishing Secrets	Jim Heid	0764540033	Paperback	626	IDG Books Worldwide	May 1994
Windows 98 Bible	Alan Simpson	0764531921	Paperback	1112	IDG Books Worldwide	June 1994
JavaScript for the World Wide Web	Tom Negrino ,Dori Smith	0201696487	Paperback	208	Peachpit Press	January 1994
HTML for the World Wide Web	Elizabeth Castro	020168862X	Paperback	255	Peachpit Press	February 1994
About Face : The Essentials of User Int	Alan Cooper	1568843224	Paperback	400	IDG Books Worldwide	August 1994
Algorithms in C	Robert Sedgewick	0201514257	Hardcover	657	Addison-Wesley	April 1994
C : A Reference Manual	Samuel P. Harbison ,Guy L	0133262243	Paperback	455	Prentice Hall Compute	December 1994
Teach Yourself C in 21 Days	Peter G. Aitken	0672310694	Paperback	736	Sams Publishing	August 1994
Teach Yourself Advanced C in 21 Days	Bradley L. Jones ,Gregory	0672304716	Paperback	878	Sams Publishing	April 1994
HTML Quick Reference	Robert Mullen	0789708671	Paperback	222	Que	1996
The Design of Everyday Things	Donald A. Norman	0385267746	Paperback	257	Currency/DoubleDay	March 1994

23 visible / 23 total

In this database we've created a form called [Detail](#) that displays the information from a single record in the database.



This form doesn't display the information directly from the database fields ([Title](#), [Authors](#), [Publisher](#), etc.). Instead it has been set up to display information from a series of variables with the same name as the fields but with an **x** added to the beginning ([xTitle](#), [xAuthors](#), [xPublisher](#), etc.). These variables are created in this procedure which opens the clone form, creates the variables and fills them with the data from the database fields.

```
local dRect
/* open the clone window */
dRect=rectanglesize(
    100+rtop(info("windowrectangle")),200+rleft(info("windowrectangle")),340,520)
setwindowrectangle dRect,"noVertScroll noHorzScroll nopalette"
fitwindow
openform "Detail"
/* create new variables for THIS window */
windowglobal xTitle,xAuthors,xISBN,xPages,xBinding,xPublisher,xDescription
/* fill the variables with the data from the current record */
xTitle=Title
xAuthors=Authors
xISBN=ISBN
xPages=Pages
xBinding=Binding
xPublisher=Publisher
xDescription=Description
/* display the variables */
showvariables xTitle,xAuthors,xISBN,xPages,xBinding,xPublisher,xDescription
```

Let's see how this procedure works. Start by selecting a record in the data sheet.

	Title	Authors	ISBN	Binding	Pages	Publisher	Edition
	Javascript Bible	Danny Goodman	0764531883	Paperback	607	IDG Books Worldwide	March 1994
	Danny Goodman's Applescript Handbook	Danny Goodman	0679758062	Paperback	554	Random House	1994
	HTML & Web Publishing Secrets	Jim Heid	0764540033	Paperback	626	IDG Books Worldwide	May 1994
	Windows 98 Bible	Alan Simpson	0764531921	Paperback	1112	IDG Books Worldwide	June 1994
	JavaScript for the World Wide Web	Tom Negrino, Dori Smith	0201696487	Paperback	208	Peachpit Press	January 1994

Now select the procedure from the **Action** menu (we have called it **Open Clone**, but you can call it anything you want). The procedure will open a new window which displays the information for **Danny Goodman's Applescript Handbook**.

Title
Javascript Bible
Danny Goodman's Applescript Handbook
HTML & Web Publishing Secrets
Windows 98 Bible
JavaScript for the World Wide Web
HTML for the World Wide Web
About Face: The Art of User Interface Design
Algorithms in C++
C: A Reference Manual
Teach Yourself C++
Teach Yourself C++
HTML Quick Reference
The Design of Everyday Things

23 visible / 23 total

ProVUE Library Clones:Detail

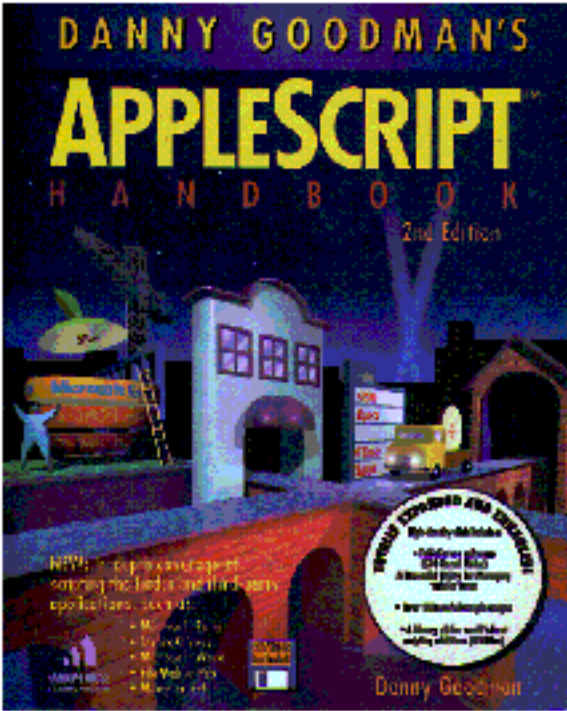
Danny Goodman's Applescript Handbook

Author : Danny Goodman

Publisher : Random House

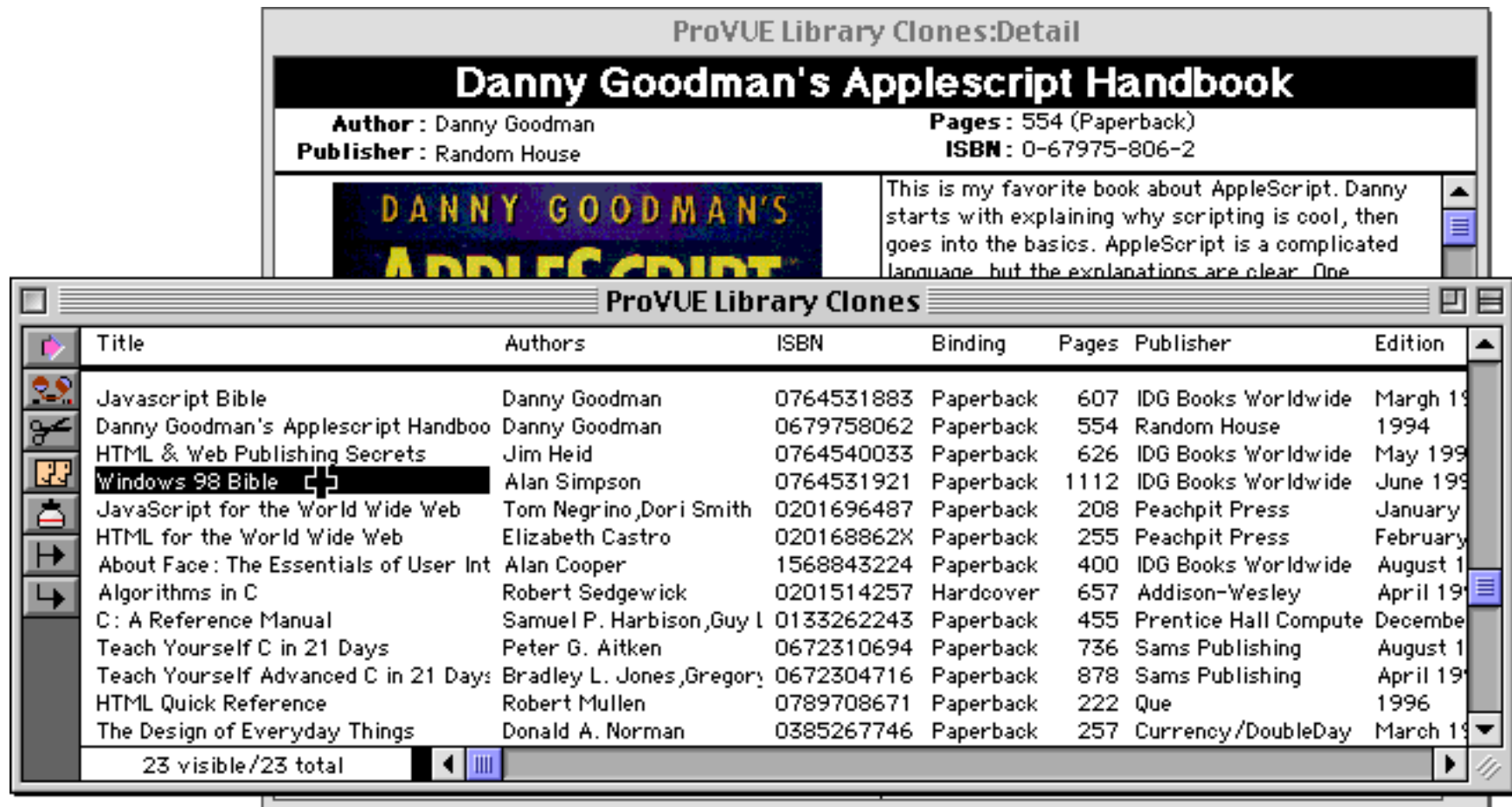
Pages : 554 (Paperback)

ISBN : 0-67975-806-2

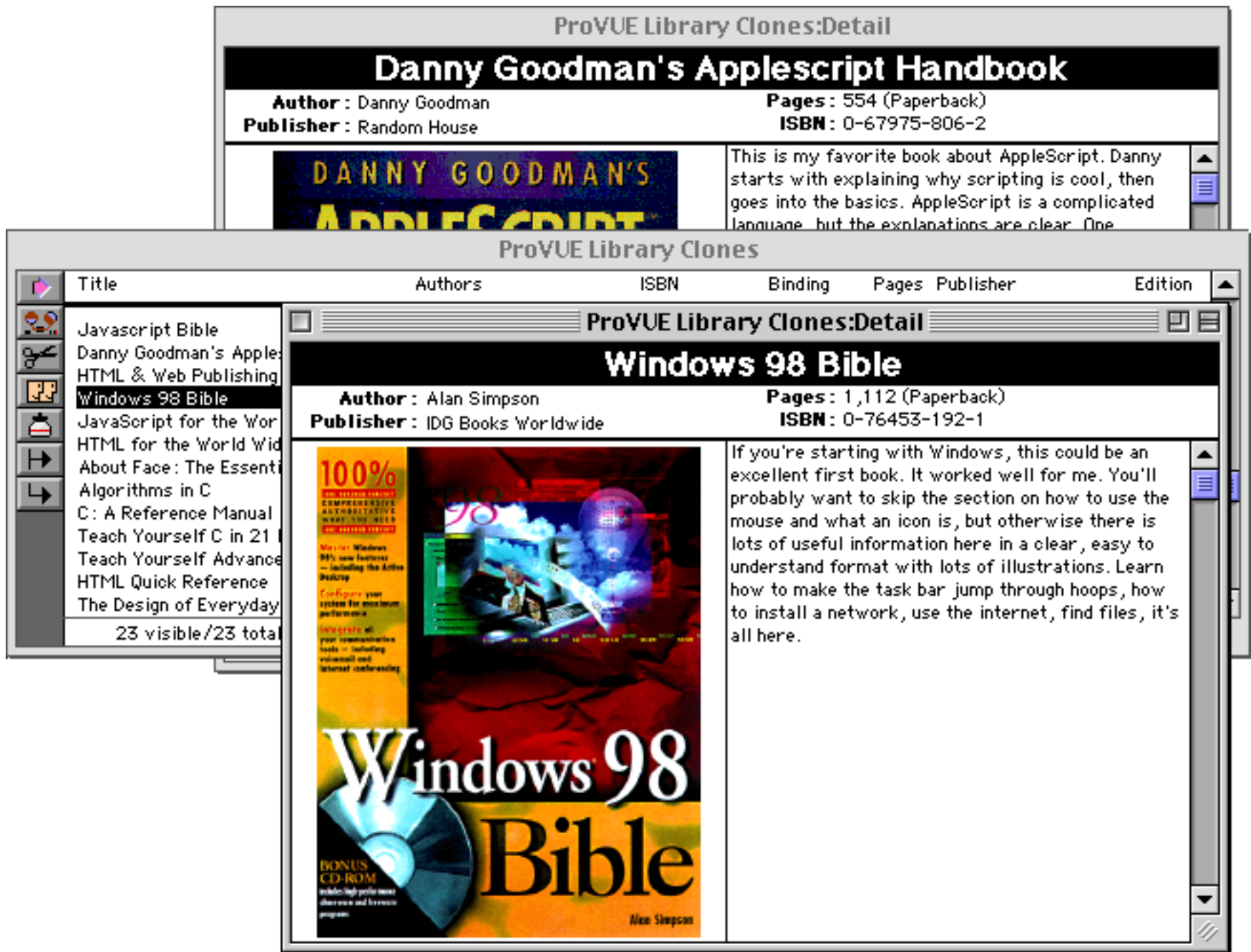


This is my favorite book about AppleScript. Danny starts with explaining why scripting is cool, then goes into the basics. AppleScript is a complicated language, but the explanations are clear. One problem is that AppleScript is really not one language, but a collection of related languages (each scriptable application is really it's own dialect). This book tackles this problem head on, showing how a number of popular languages are scripted, and giving tools for figuring out how to script a new application (probably with lousy AppleScript documentation). Amazon says this book is out of print, but if you can find it, grab it.

Now click on the data sheet to bring it back to the front and then click on another record. Usually when you do this any open forms will automatically synchronize to show the new record. But in this case, the form is not displaying the information directly from the database but instead is displaying the data we have stored in the `windowGlobal` variables. These variables have not changed, so the form continues to display the data from the original record.



Now if we select the **Open Clones** procedure again the procedure will open a second copy of the **Detail** form. This new copy shows the information from the current record, while the original **Detail** window continues to show the information from the original window.



Using the **Open Clone** procedure we can continue to open additional “clone” copies of the form — as many as we want up to Panorama’s 32 window limit.



There is no special handling necessary for closing clone windows. The window simply closes when you click on the close box. All of the window global variables associated with the window are destroyed when the window is closed.

Alerts

Alerts are very simple dialogs that simply display a message and allow the user to press a button. Alerts are usually used to “alert” the user of a situation (a problem, perhaps). Panorama has several off-the-shelf alerts. (You can also create your own alerts with a form, just like any other dialog.)

The simplest way to alert the user to a situation is to use the **beep** statement. This statement, which has no parameters, simply causes the computer to make its standard beep sound.

Displaying a Message

The simplest way to display a short piece of text is with the **message** statement. This simply displays an alert with any message you want. The alert stays on the screen until the user presses the **OK** button. The message statement has one parameter, the text of the message to be displayed. The example below displays the number of records in the database.

```
message "Total records: "+str(info("records"))
```

Here is what this alert looks like when this procedure runs.



There are also three variations on the `Message` statement — `BigMessage`, `GiantMessage` and `TallMessage`. If you use these, keep in mind that you can only display 256 characters in these dialogs. So the `TallMessage` statement, for example, is useful for a narrow list of items, but you definitely can't fill the dialog with text (the `DisplayData` and `SuperAlert` statements described below will allow that).



If you really want to display a lot of text use the `DisplayData` statement. For example, this program will ask you to select a file, then display the entire file.

```
local folder,file,type,filetext
openfiledialog folder,file,type," "
if file="" rtn endif
filetext=fileload(folder,file)
displaydata filetext
```

This statement will allow you to display up to 32,768 characters. The dialog normally takes up most of whatever screen size you have available (the illustration below has been reduced to 60%) and has a scroll bar so that you can view additional text.

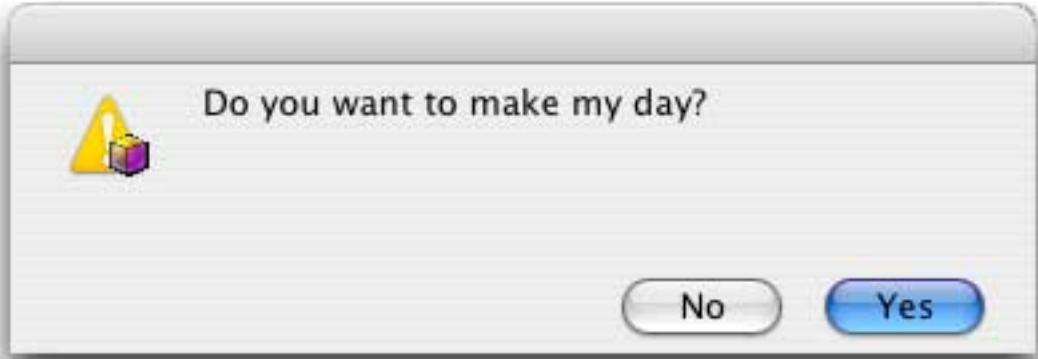


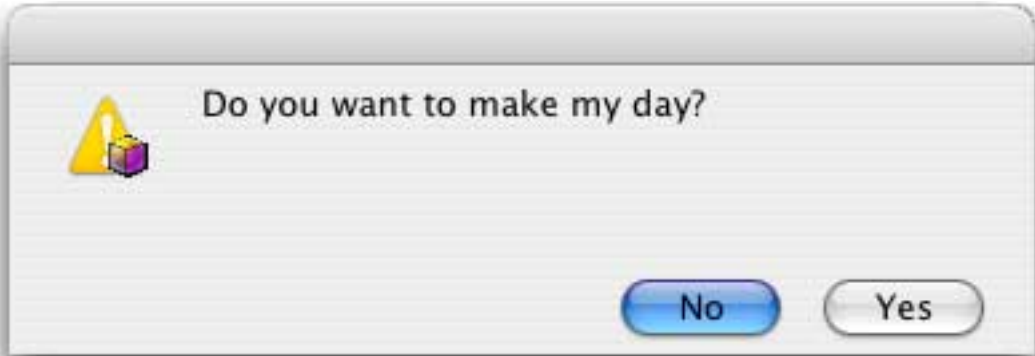
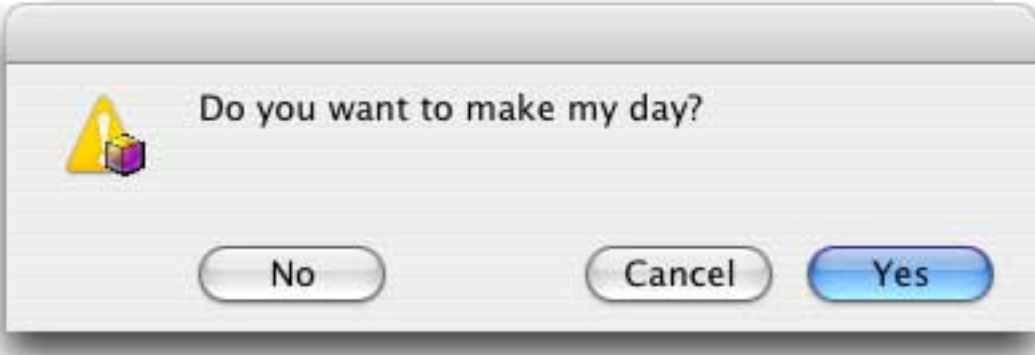
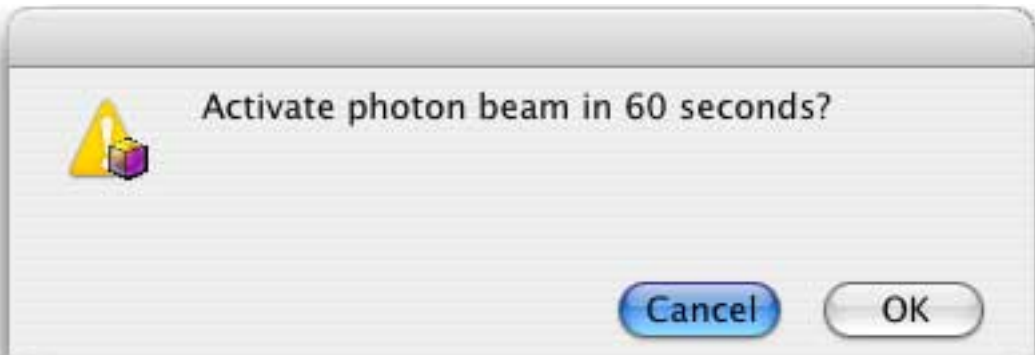
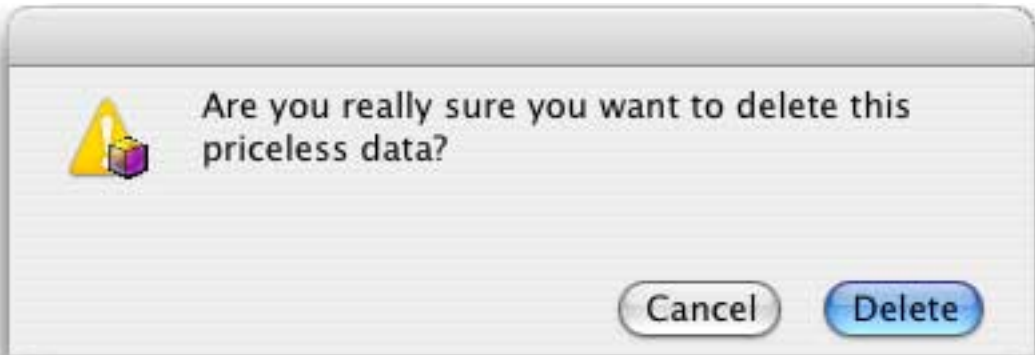
There are three buttons at the bottom of the dialog. To simply continue, press the **OK** button. To stop the procedure, press the **Stop** button. Press the **Copy** button to copy the text that is displayed to the clipboard (this also stops the procedure).

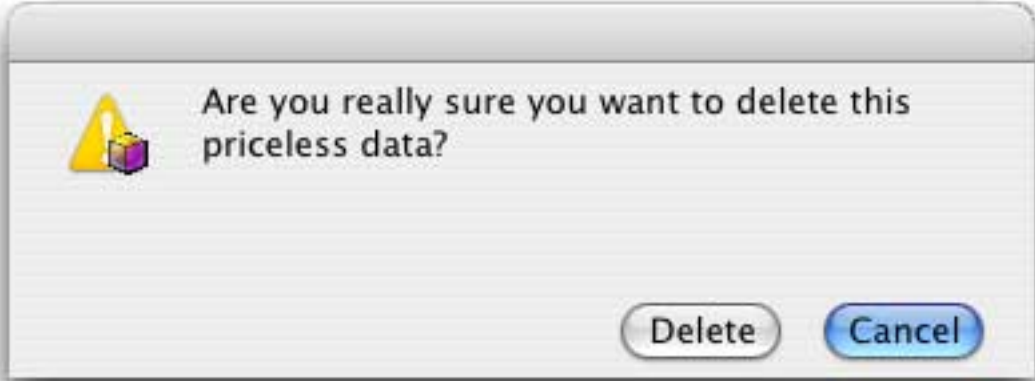
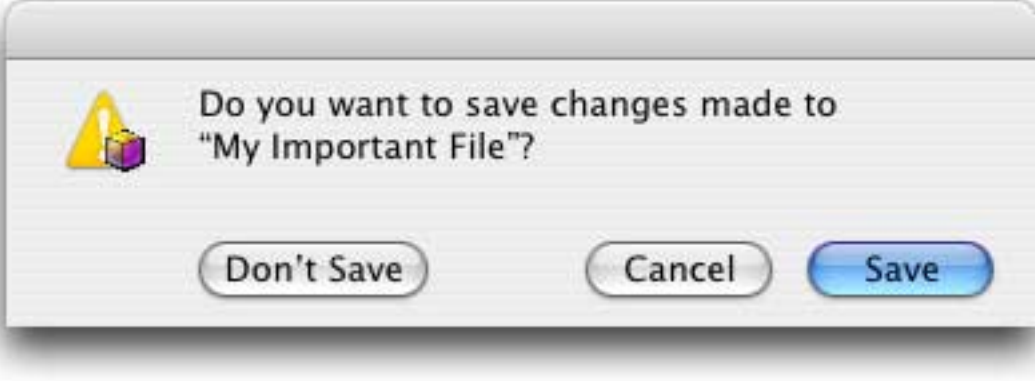



The `displaydata` statement has an optional second parameter that allows you to customize the appearance of the dialog. You can change the size, title, font, text size, color and many other options. For more information see "[The SuperAlert Statement](#)" on page 472.

Alerts With Multiple Buttons

Panorama has several statements for creating standard alerts with multiple buttons. After this statement the program can use the `info("dialogtrigger")` function to find out what button was pressed. This function will return the text of the button, for example, **Yes**, **No**, **Cancel**, etc.

Code	Sample
<pre>alertyesno "Do you want to make my day?"</pre>	

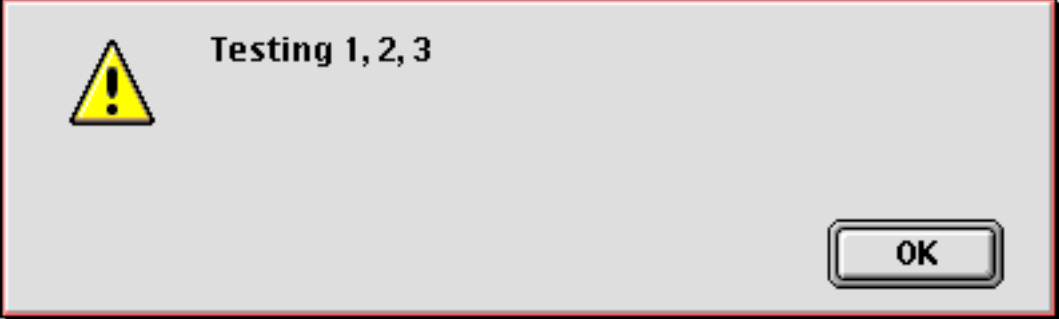
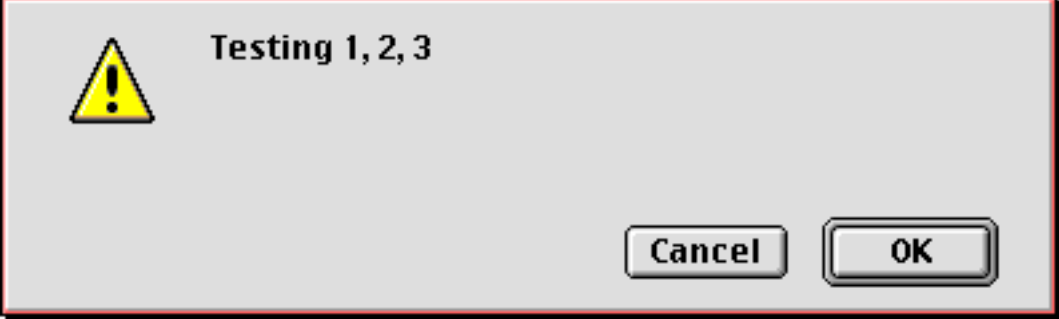
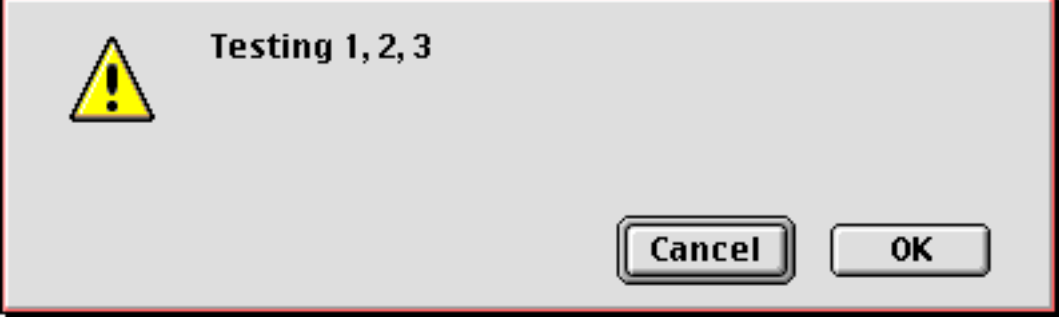
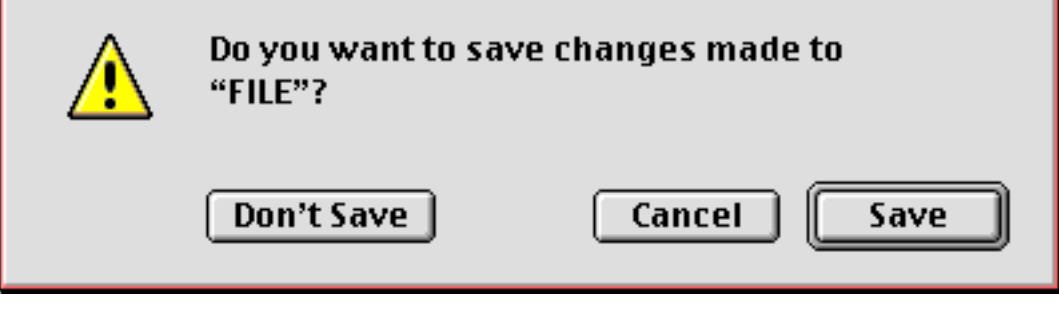
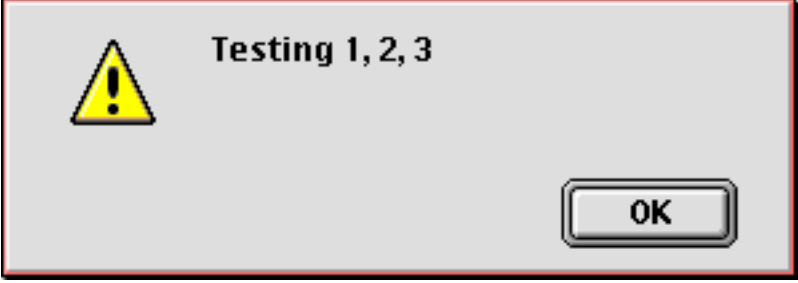
Code	Sample
<pre>alertnoyes "Do you want to make my day?"</pre>	 An alert dialog box with a yellow warning triangle icon and a gift icon. The text reads "Do you want to make my day?". At the bottom right, there are two buttons: "No" (highlighted in blue) and "Yes".
<pre>alertyesnocancel "Do you want to make my day?"</pre>	 An alert dialog box with a yellow warning triangle icon and a gift icon. The text reads "Do you want to make my day?". At the bottom, there are three buttons: "No", "Cancel", and "Yes" (highlighted in blue).
<pre>alertokcancel "Activate photon beam in 60 seconds?"</pre>	 An alert dialog box with a yellow warning triangle icon and a gift icon. The text reads "Activate photon beam in 60 seconds?". At the bottom right, there are two buttons: "Cancel" and "OK" (highlighted in blue).
<pre>alertcancellook "Activate photon beam in 60 seconds?"</pre>	 An alert dialog box with a yellow warning triangle icon and a gift icon. The text reads "Activate photon beam in 60 seconds?". At the bottom right, there are two buttons: "Cancel" (highlighted in blue) and "OK".
<pre>alertdeletecancel "Are you really sure you want to delete this priceless data?"</pre>	 An alert dialog box with a yellow warning triangle icon and a gift icon. The text reads "Are you really sure you want to delete this priceless data?". At the bottom right, there are two buttons: "Cancel" and "Delete" (highlighted in blue).

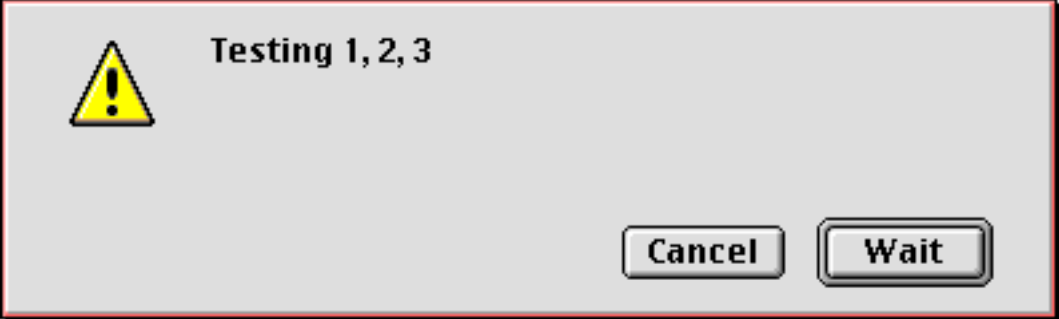
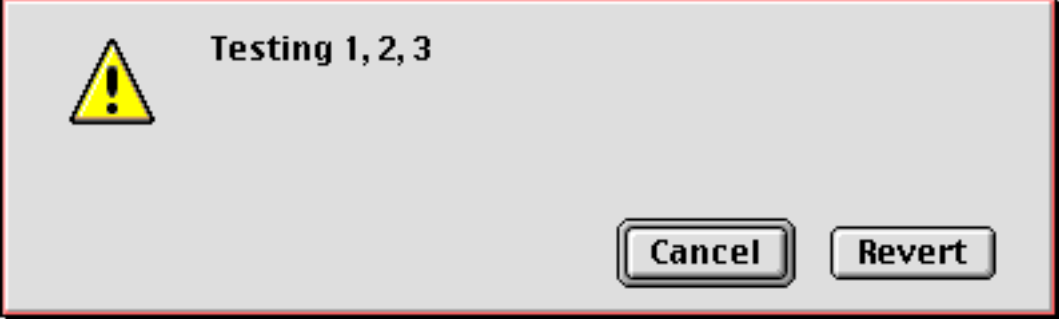
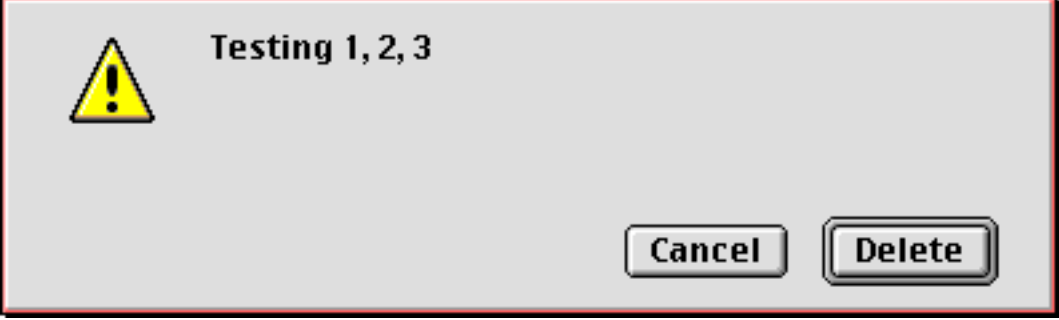
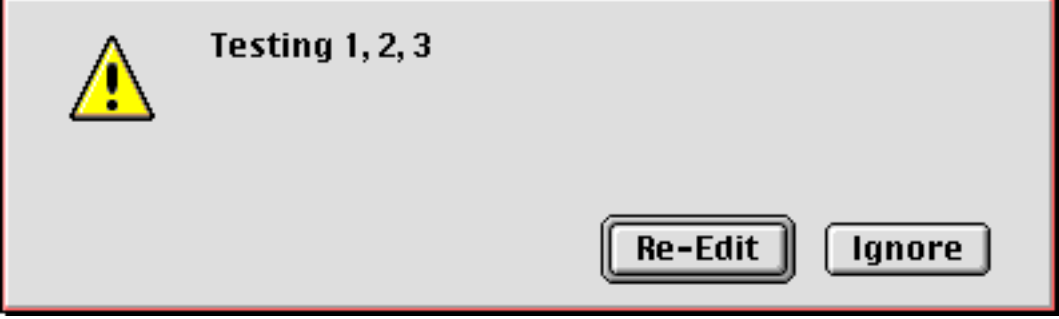
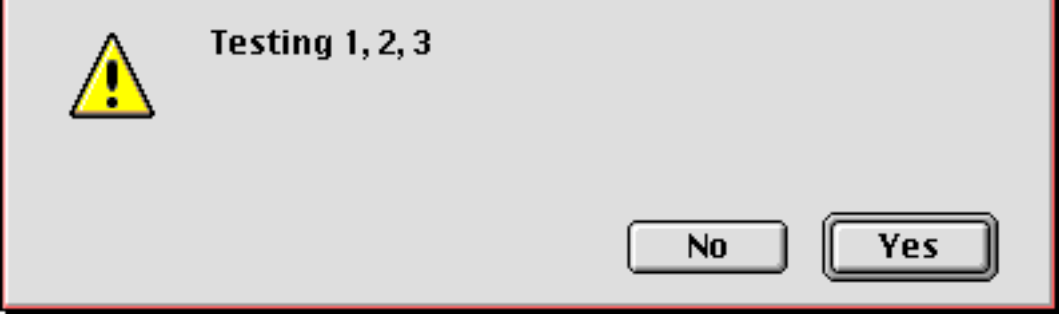
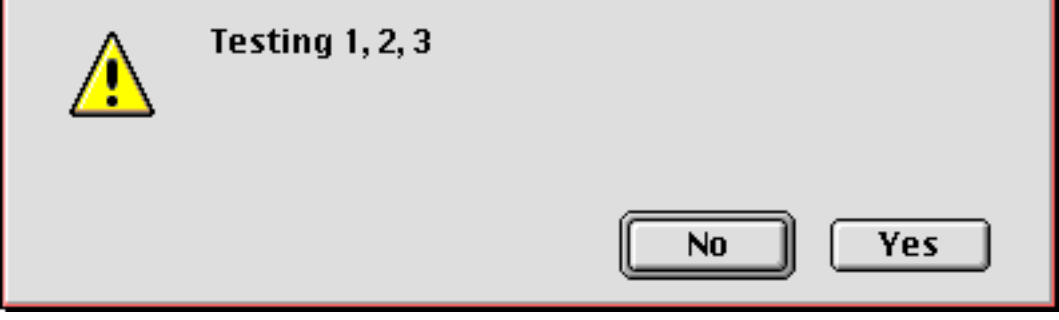
Code	Sample
<pre>alertcanceldelete "Are you really sure you want to delete this priceless data?"</pre>	
<pre>alertsavcancel "My Important File"</pre>	
<pre>alertrevertcancel "Do you want to revert to the previous version?"</pre>	
<pre>alertok "This is identical to the message statement except for the icon."</pre>	
<pre>alertoksmall "When you have very little to say, this one is perfect!"</pre>	


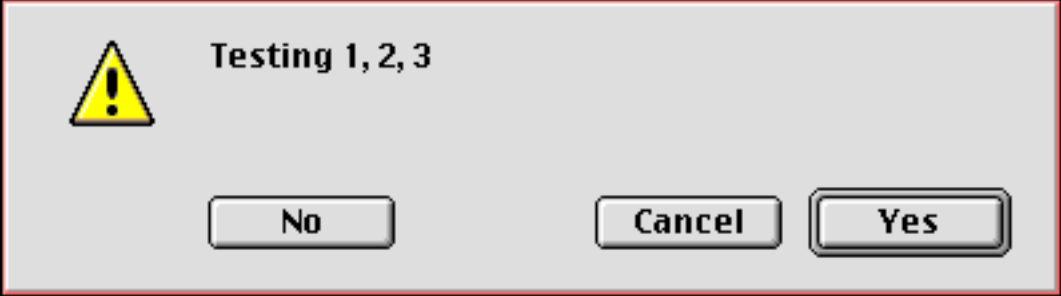
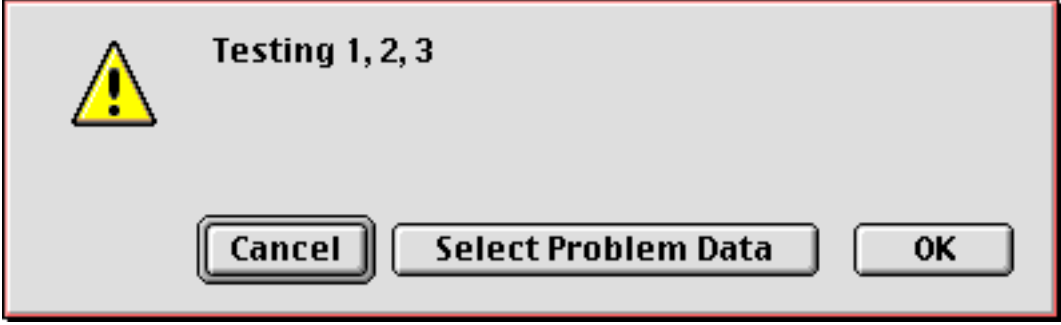
The Alert Statement

The `alert` statement allows you to build your own custom alerts with **ResEdit** (see “[Working with Resources](#)” on page 433). However, we no longer recommend this statement — use the **SuperAlert** statement instead (see “[The SuperAlert Statement](#)” on page 472). The `alert` statement has two parameters: `id` and `message`. The `id` is the resource number of the alert template you have built. The `message` is the text you want to display in the alert. Like the statements above, to find out what button was pressed use the `info("dialogtrigger")` function.

There are several alert templates built into Panorama that you can use with the alert statement.

Code	Sample
<pre>alert 1000,"Testing 1, 2, 3"</pre>	
<pre>alert 1001,"Testing 1, 2, 3"</pre>	
<pre>alert 1002,"Testing 1, 2, 3"</pre>	
<pre>alert 1003,"FILE"</pre>	
<pre>alert 1005,"Testing 1, 2, 3"</pre>	

Code	Sample
<pre>alert 1008,"Testing 1, 2, 3"</pre>	
<pre>alert 1009,"Testing 1, 2, 3"</pre>	
<pre>alert 1010,"Testing 1, 2, 3"</pre>	
<pre>alert 1012,"Testing 1, 2, 3"</pre>	
<pre>alert 1013,"Testing 1, 2, 3"</pre>	
<pre>alert 1014,"Testing 1, 2, 3"</pre>	

Code	Sample
<pre>alert 1015,"Testing 1, 2, 3"</pre>	
<pre>alert 1018,"Testing 1, 2, 3"</pre>	
<pre>alert 1101,"Testing 1, 2, 3"</pre>	

The example below uses the `alert` statement instead of the `noyes` statement.

```
alert 1014,"Do you really want to remove the old data?"
if info("dialogtrigger")="Yes"
  select Date>today()-90
  removeunselected
endif
```

If you create your own alert templates with **ResEdit**, make sure the resource file is open (use the **openresource** statement) before you attempt to use the alert (see “[Opening and Closing Resource Files](#)” on page 435).

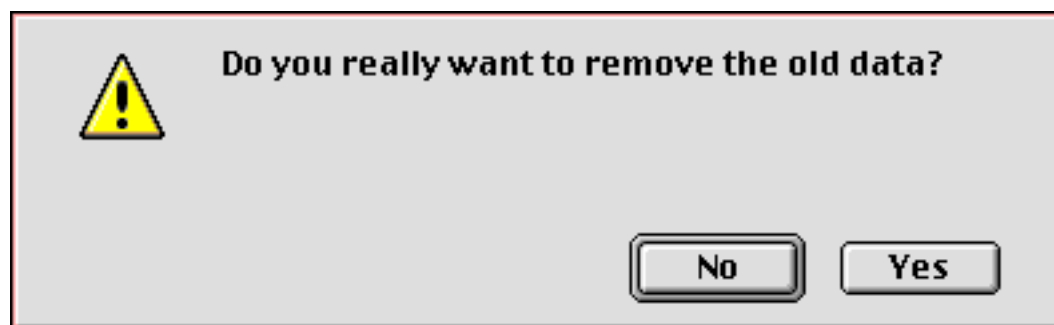
Obsolete Alert Statements

Since Panorama was first released in 1988 it has evolved and improved in many ways. For compatibility, we usually leave in older features even when new features make them obsolete, as is the case with the statements described in this section. However, they are still described here because you may encounter them in older databases. These statements display a message and allow the user to make a choice: **Yes** or **No**, **Ok** or **Cancel**, etc. The statements that display these alerts are `yesno`, `noyes`, `okcancel`, and `cancelok`. These statements all display an alert with two buttons. For example, the `yesno` statement displays a dialog with **Yes** and **No**

buttons. Notice that the first button is the default button, so the difference between `yesno` and `noyes` is which button is the default. All of these statements have one parameter: the text of the message to be displayed. The statements will put the name of the button clicked into the clipboard. The example below uses the `noyes` statement to confirm that the user really wants to delete data from the database.

```
noyes "Do you really want to remove the old data?"
if clipboard()="Yes"
  select Date>today()-90
  removeunselected
endif
```

When this procedure is run the alert looks like this.



Using the newer `alertnoyes` statement, this procedure would look like this:

```
alertnoyes "Do you really want to remove the old data?"
if info("dialogtrigger")="Yes"
  select Date>today()-90
  removeunselected
endif
```

The advantage of the `alertnoyes` statement is that it doesn't disturb the clipboard.

Suppressing Alerts

In some applications (particularly web servers) you may want to suppress all alerts so that the program never stops and waits for someone to do something. You can do this with the `alertmode` statement. This statement has one parameter, which controls whether alerts will appear. If the parameter is "yes", "true", or "on", alerts will be displayed. If the parameter is "no", "false", or "off", alerts will not be displayed. The program will simply continue as if the default button had been pressed.

The SuperAlert Statement

Panorama V introduced this new statement that displays a configurable alert. You can control over a dozen different options, including the overall alert size (height and width), the font, text size, title, color, background color, and more. You can even place images on the alert and specify up to three buttons on the alert. In addition, the alert may display up to 32,768 characters (it is *not* limited to 255 characters like the other alerts described in this section) and can even have a scroll bar.

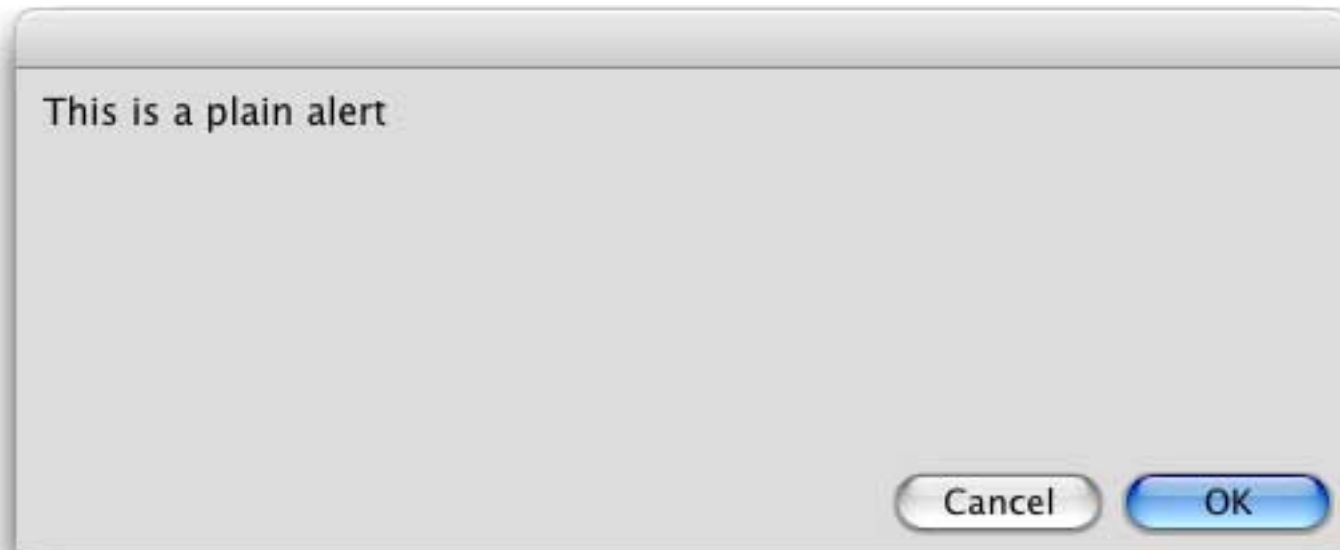
The SuperAlert statement has two parameters

```
superalert message,options
```

Message is simply whatever text you want to display in the alert. **Options** is a text parameter that uses a syntax similar to an HTML tag to specify one or more options. If you just want to use the default options you don't have to specify any options at all.

```
superalert "This is a plain alert", ""
```


This plain alert looks like this:



Each option is specified as an **option=value** pair, for example **height=4in**, **color=red**, etc. Here is an example of a dialog with several options set. (Tip: If you enclose the entire option parameter in “smart quotes” as shown below you will be able to use regular " quotes for the individual options, if necessary.)

```
superalert "WARNING!",  
    "height=4in font=helvetica size=36 color=red style=bold textalign=center buttons=Danger"
```

This alert should get some attention!



The table below describes each of options available with this statement.

Option	Examples	Description
title=	title="Latest Information"	With this option you can set the title of the alert. This appears in the title bar at the top of the alert window.
height=	height=400 height=5in height=4.5" height=10cm height=80% height=-50 height=-2cm	This option specifies the height of the alert. The default is 200 pixels. If you supply just a number then the height is specified in pixels (72 pixels = 1 inch). If the number is followed by in or " then it is specified in inches. If the number is followed by cm it is specified in centimeters. If the number is followed by % it is specified as a percentage of the screen height, for example 50% is 1/2 of the screen height. If the height is negative then this value is the distance of the border between the top of the screen and the alert (also the bottom of the alert and the bottom of the screen). For example if your screen is 8 inches high then a height of -1in will make the alert 6 inches high (one inch at the top and one inch at the bottom).
width=	width=600 width=7in width=5.5" width=12cm width=80% width=-50 width=-2cm	This option specifies the width of the alert. The default is 500 pixels. If you supply just a number then the width is specified in pixels (72 pixels = 1 inch). If the number is followed by in or " then it is specified in inches. If the number is followed by cm it is specified in centimeters. If the number is followed by % it is specified as a percentage of the screen width, for example 50% is 1/2 of the screen width. If the width is negative then this value is the distance of the border between the left of the screen and the alert (also the right edge of the alert and the right side of the screen). For example if your screen is 10 inches wide then a width of -1in will make the alert 8 inches wide (one inch on the left and one inch on the right).
font=	font=Tekton font="Comic Sans" font=Verdana	This option specifies the font to be used. The default is whatever the system font is on your system.
size=	size=9 size=18	This option specifies the text size (in points). The default is whatever the default system text size is on your system.
style=	style=italic style=bold style="bold italic"	This option specifies the text style. You can choose one or more options from bold, italic, underline, outline and shadow. All of the text in the alert will be displayed in the same style, you cannot mix styles within the alert.
textalign=	textalign=center	This option specifies how the text will be positioned within the alert. The default is <i>topleft</i> . The available choices are: <i>topleft</i> , <i>topcenter</i> , <i>topright</i> , <i>leftcenter</i> , <i>center</i> , <i>rightcenter</i> , <i>bottomright</i> , <i>bottomcenter</i> , and <i>bottomleft</i> .
scroll=	scroll=yes scroll=thin	This option enables a scroll bar to allow viewing of large quantities of text. The options are <i>yes</i> , <i>thin</i> and <i>no</i> .
buttons=	buttons="Yes;No;Cancel" buttons="One:50;Two:50"	This option allows you to specify up to three buttons (the default is two buttons: <i>Ok</i> and <i>Cancel</i>). Each button is separated by a semicolon, and the first button listed is the default button. The default button width is 80 pixels, you can also specify a button width in pixels by placing a colon followed by the width after the button name. In the second example to the left the buttons will be 50 pixels wide.
timeout=	timeout=20	If this option is used the alert will close automatically after the specified number of seconds. If the user doesn't press a button before the specified time the alert will automatically press the default button so the procedure can continue.

Option	Examples	Description
color=	<pre>color=#00FF00 color=red color=royalblue</pre>	<p>This option specifies the text color (the default is black). If the color begins with # then you can specify any color using HTML style color tags (#RRGGBB, for example #00FF00 for green). You can also choose from this list of colors: <i>aliceblue, antiquewhite, aqua, aquamarine, azure, beige, bisque, black, blanchedalmond, blue, blueviolet, brown, burlywood, cadetblue, chartreuse, chocolate, coral, cornflowerblue, cornsilk, crimson, cyan, darkblue, darkcyan, darkgoldenrod, darkgray, darkgreen, darkkhaki, darkmagenta, darkolivegreen, darkorange, darkorchid, darkred, darksalmon, darkseagreen, darkslateblue, darkslategray, darkturquoise, darkviolet, deeppink, deepskyblue, dimgray, dodgerblue, firebrick, floralwhite, forestgreen, fuchsia, gainsboro, ghostwhite, gold, goldenrod, gray, green, greenyellow, honeydew, hotpink, indianred, indigo, ivory, khaki, lavender, lavenderblush, lawngreen, lemonchiffon, lightblue, lightcoral, lightcyan, lightgoldenrodyellow, lightgreen, lightgrey, lightpink, lightsalmon, lightseagreen, lightskyblue, lightslategray, lightsteelblue, lightyellow, lime, limegreen, linen, magenta, maroon, medianaquamarine, mediumblue, mediumorchid, mediumpurple, mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise, mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, navajowhite, navy, oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen, paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, powderblue, purple, red, rosybrown, royalblue, saddlebrown, salmon, sandybrown, seagreen, seashell, sienna, silver, skyblue, slateblue, slategray, snow, springgreen, steelblue, tan, teal, thistle, tomato, turquoise, violet, wheat, white, whitesmoke, yellow, yellowgreen.</i></p>

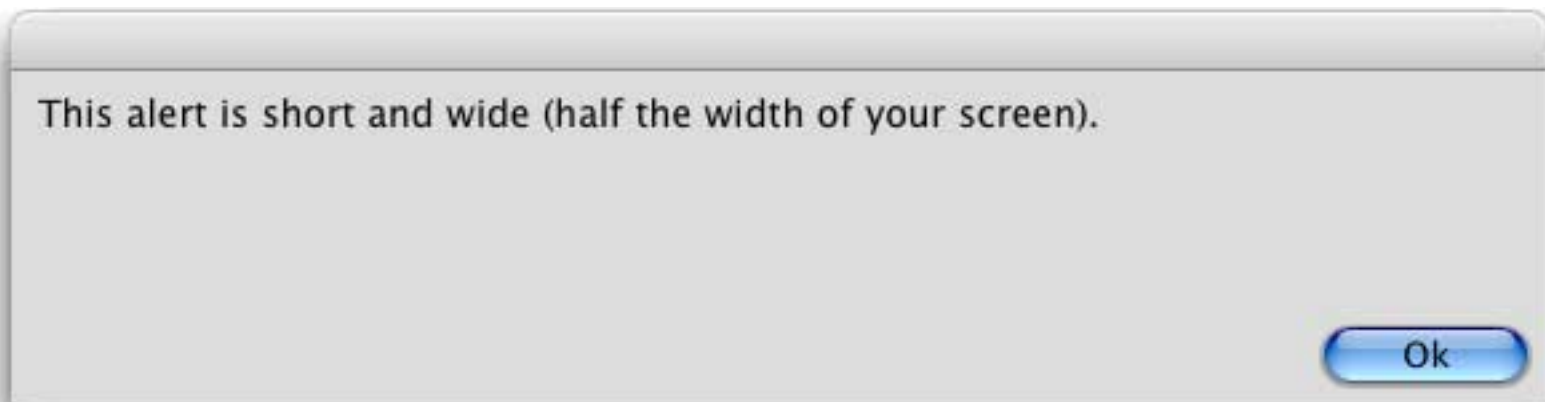
Option	Examples	Description
bgcolor=	bgcolor=#FFDDFF bgcolor=lightskyblue bgcolor=##983	This option specifies the background color (the default is gray). If the color begins with # then you can specify any color using HTML style color tags (#RRGGBB, for example #FFCCFF for lite pink). You can also choose from the same list of colors available for the color= option (see above). A third option is to use ## followed by a resource number to display a background image from Panorama's resources. You can use the Icons & Backgrounds wizard to find out what background images are available.
icon=	icon=stop icon=caution icon=note	This option will display a stop sign, caution sign, or note icon in the alert.
image=	image="Smiley Face"	This option allows you to display any image you want within the alert. The image may be in a disk file or in the flash art gallery of the current database. If the image is in a disk file the default location is the folder containing the current database, but you can specify any folder you want by specifying the complete path.
imageedge=	imageedge=top imageedge=left imageedge=both imageedge=none	This option specifies how the image should be placed relative to the text. <i>Left</i> means that the image will be placed to the left of the text. <i>Top</i> means that the image will be placed above the text. <i>Both</i> means the text will be placed both below and to the right of the image (the image in the upper left corner with the text in the lower right corner). <i>None</i> means that the text will be drawn on top of the image, essentially making the image a background. For all of these options the alert will adjust the text location based on the size of the image.
imagealign=	imagealign=topcenter imagealign=leftcenter	This option specifies how the image should be aligned relative to the edges of the alert. The options are <i>topleft</i> (the default), <i>topcenter</i> , <i>topright</i> , <i>leftcenter</i> and <i>bottomcenter</i> .
texttopadjust=	texttopadjust=10	The text position is automatically adjusted based on the image size. If the image is placed at the top (imageedge=top) the text is normally placed 3 pixels below the image. This option allows you to change that spacing.

Here are some examples illustrating the use of the `superalert` statement.

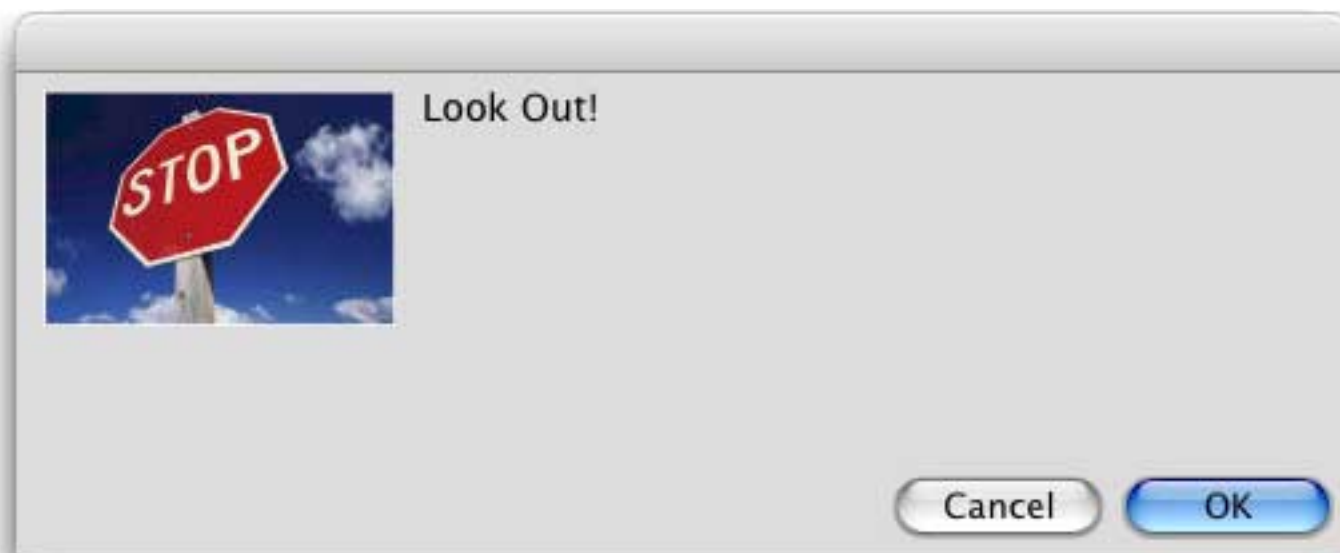
```
superalert "Are you talking to me?",  
  {height=120 width=200 font="Verdana" size=9 color="red" buttons="Yes:60;No:60"}
```



```
superalert "This alert is short and wide (half the width of your screen).",  
  {height=2in width=50% buttons="Ok"}
```



```
superalert "Look Out!", {image="Stop Sign"}
```



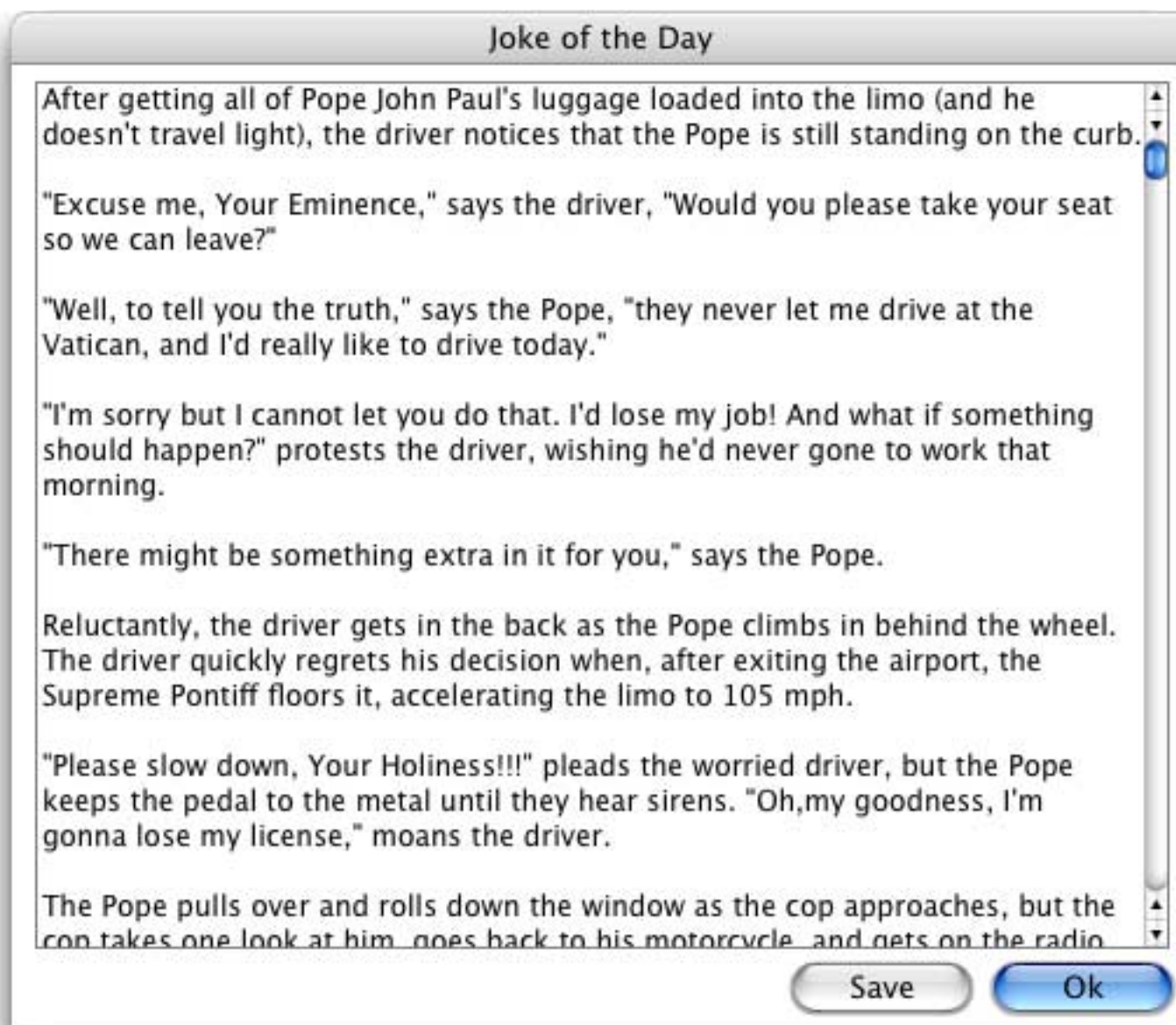
```
superalert "Look Out!",{image="Stop Sign" imageedge=top bgcolor=white}
```



```
superalert "Look Out!",{image="Stop Sign" imagealign=topcenter imageedge=top bgcolor=white}
```



```
superalert Joke,"Title="Joke of the Day" height=6in width=7in scroll=thin size=12
bgcolor=white buttons="Ok;Save"
```



The DisplayData Alert

The `DisplayData` alert (see “[Displaying a Message](#)” on page 464) is based on the `SuperAlert` statement. The second parameter to the `DisplayData` statement can take all of the same options as the `SuperAlert` statement, but has different default values. You can override these defaults by supplying a second parameter with the options you want changed.

```
displaydata myFile,"title="My File" font=Courier size=12"
```

The default options are white background, Monaco 9 point font, window height 100 pixels less than screen height, width 80% of screen width, with three buttons: **Ok**, **Stop** and **Copy**. (You can eliminate the **Stop** and/or **Copy** buttons, but if you add other buttons they will not work. Use the `superalert` statement directly if you need to add other buttons.)

Dialogs

Dialogs are a special type of window. A dialog window is usually temporary, and usually modal. In other words, the dialog must be filled in and dismissed before the user can continue with his or her work.

Basic Text Entry Dialogs

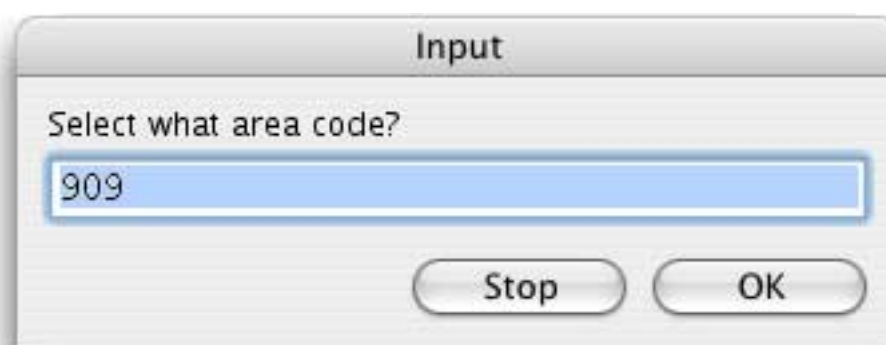
The most basic dialog prompts for a single item of text. Panorama has several options for easily creating this type of basic dialog, starting with the `gettext` statement. This statement has two parameters.

```
gettext prompt,input
```

The `prompt` parameter should be a short message that will be displayed in the dialog. This message should explain what needs to be entered. The `input` parameter must be the name of a field or variable that contains text. The value in this field or variable will be displayed in the dialog—the user can use it as is, edit the value, or erase it completely and type in a new value. The example below uses `gettext` to find out what area code to search for. The default area code is **909**.

```
local whatArea
whatArea="909"
gettext "Select what area code?",whatArea
select Phone match "("+whatArea+")*"
```

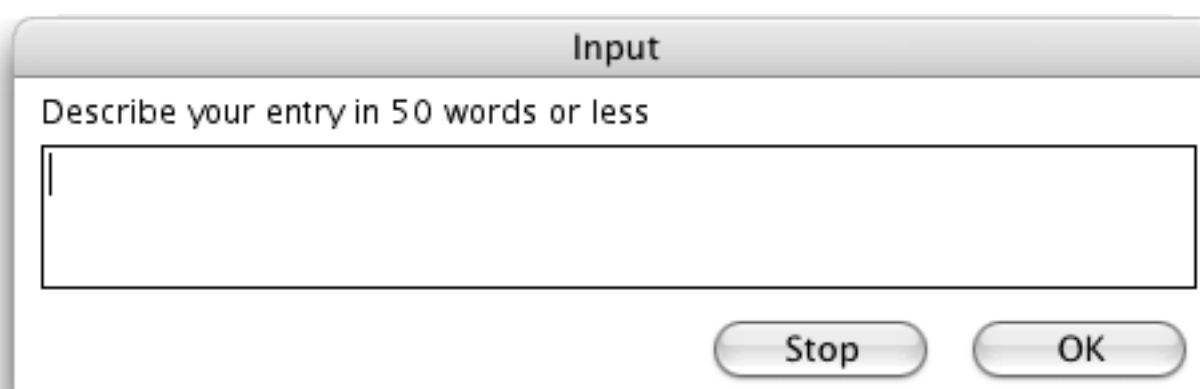
When this procedure is run a dialog like this is displayed.







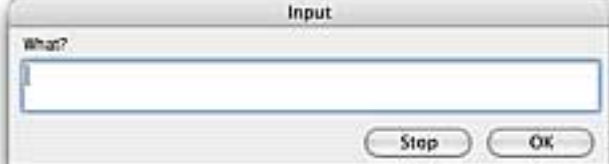
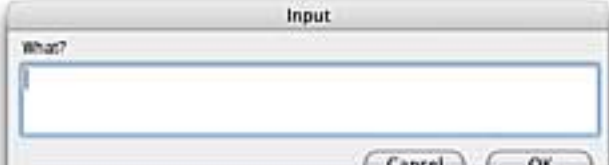



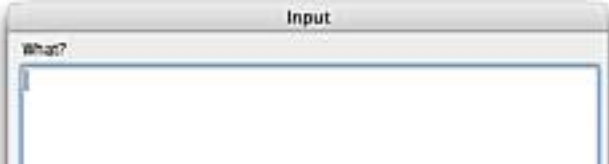
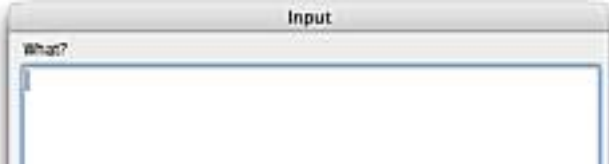
You can customize the appearance of this basic dialog by using the `customdialog` statement. This statement has one parameter, the resource ID number of a dialog template. You can create dialog templates with **ResEdit** (see [“Working with Resources”](#) on page 433), or you can use one of several templates supplied with Panorama. Here’s a procedure that uses one of Panorama’s built in templates.

```
customdialog 3103
gettext "Describe your entry in 50 words or less",Description
```

Here is the alert that will appear.



This table shows the different templates that are available as part of Panorama. (To save space, these images have been reduced to 50%).

#	Sample	#	Sample
3131			
3121		3101	
3122		3102	
3123		3103	
3125		3105	
3120		3100	

As you may have noticed, some of these dialogs contain a **Cancel** button and some contain a **Stop** button. If the dialog contains a **Stop** button the procedure will stop immediately if the button is pressed.

If the dialog contains a **Cancel** button the procedure will continue no matter what button is pressed. The procedure can use the `info("dialogtrigger")` function to find out which button was pressed. If the **Cancel** button was pressed Panorama will ignore whatever the user typed into the field or variable, leaving the original value untouched. Here is a procedure that uses the `info("dialogtrigger")` function to find out which button was pressed.

```
local whatArea
whatArea="909"
customdialog 3131
gettext "Select what area code?",whatArea
if info("dialogtrigger")="OK"
    select Phone match ("+"whatArea+)*"
else
    selectall
endif
```

If you create your own custom resource templates make sure the resource file is open before you use the dialog (see "[Opening and Closing Resource Files](#)" on page 435).

The SuperGetText Statement

Panorama V introduced this new statement that displays a configurable dialog for entering text. You can control over a dozen different options, including the overall dialog size (height and width), the font, text size, title, color, background color, and more.

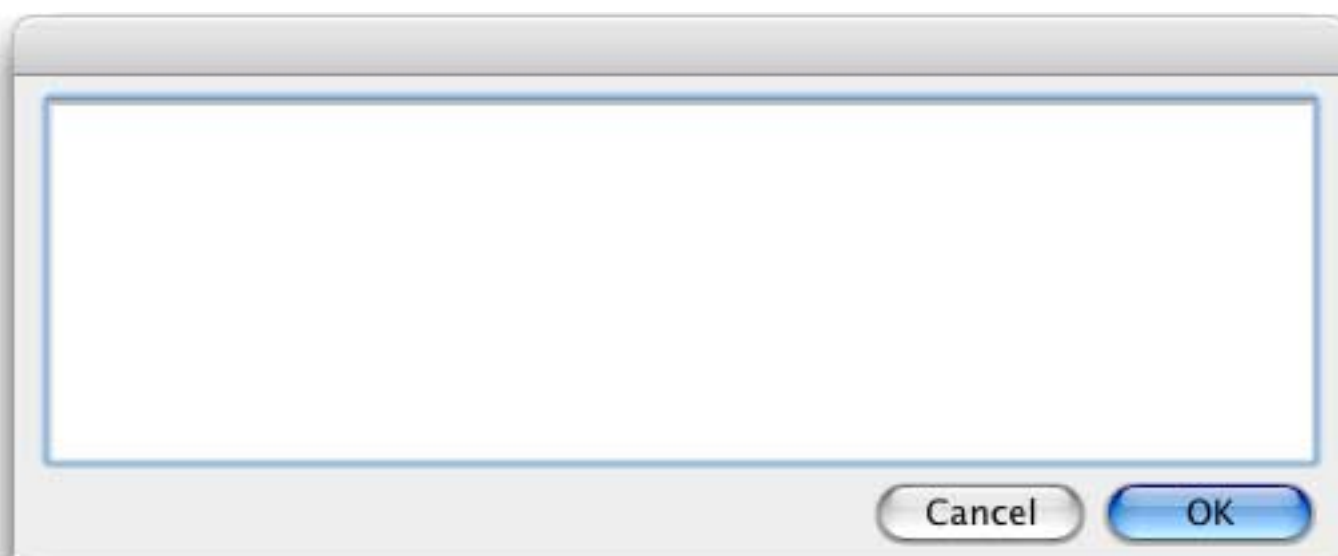
The SuperGetText statement has two parameters

```
supergettext input,options
```

Input is a field or variable that will contain the text that is typed into the dialog. **Options** is a text parameter that uses a syntax similar to an HTML tag to specify one or more options. If you just want to use the default options you don't have to specify any options at all.

```
supergettext Query,""
```

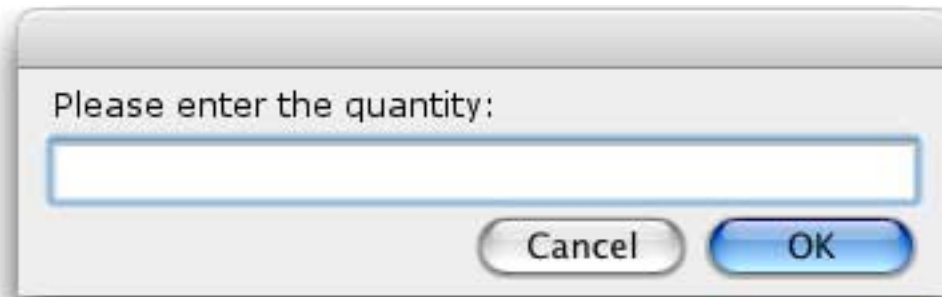
This plain dialog looks like this:



Each option is specified as an `option=value` pair, for example `height=4in`, `caption="Enter Quantity"`, etc. Here is an example of a dialog with several options set. (Tip: If you enclose the entire option parameter in "smart quotes" as shown below you will be able to use regular " quotes for the individual options, if necessary.)

```
supergettext Query,"caption="Please enter the quantity:" height=104 width=350"
```

This dialog will look like this.



The table below describes each of options available with this statement.

Option	Examples	Description
title=	title="Latest Information"	With this option you can set the title of the dialog. This appears in the title bar at the top of the dialog window.
height=	height=400 height=5in height=4.5" height=10cm height=80% height=-50 height=-2cm	This option specifies the height of the dialog. The default is 200 pixels. If you supply just a number then the height is specified in pixels (72 pixels = 1 inch). If the number is followed by in or " then it is specified in inches. If the number is followed by cm it is specified in centimeters. If the number is followed by % it is specified as a percentage of the screen height, for example 50% is 1/2 of the screen height. If the height is negative then this value is the distance of the border between the top of the screen and the alert (also the bottom of the alert and the bottom of the screen). For example if your screen is 8 inches high then a height of -1in will make the alert 6 inches high (one inch at the top and one inch at the bottom).
width=	width=600 width=7in width=5.5" width=12cm width=80% width=-50 width=-2cm	This option specifies the width of the dialog. The default is 500 pixels. If you supply just a number then the width is specified in pixels (72 pixels = 1 inch). If the number is followed by in or " then it is specified in inches. If the number is followed by cm it is specified in centimeters. If the number is followed by % it is specified as a percentage of the screen width, for example 50% is 1/2 of the screen width. If the width is negative then this value is the distance of the border between the left of the screen and the alert (also the right edge of the alert and the right side of the screen). For example if your screen is 10 inches wide then a width of -1in will make the alert 8 inches wide (one inch on the left and one inch on the right).
font=	font=Tekton font="Comic Sans" font=Verdana	This option specifies the font to be used. The default is whatever the system font is on your system.
size=	size=9 size=18	This option specifies the text size (in points). The default is 12 points.
scroll=	scroll=yes scroll=thin	This option enables a scroll bar to allow viewing of large quantities of text. The options are <i>yes</i> , <i>thin</i> and <i>no</i> .
caption=	caption="Start date:"	This option specifies a message to be displayed at the top of the dialog, above the text. (This message is displayed within the dialog, not in the title bar.)
captionheight=	captionheight=2	This option specifies how many lines should be used for the caption (see above). This value defaults to 1, but you can use a larger value if you have a long caption.

Option	Examples	Description
captionfont=	captionfont=Tekton captionfont="Comic Sans" captionfont=Verdana	This option specifies the font to be used in the caption. This allows you to specify a different font for the caption and the text editing portions of the dialog.
captionsize=	captionsize=9 captionsize=18	This option specifies the text size (in points) to be used in the caption. This allows you to specify a different font size for the caption and the text editing portions of the dialog.
captionstyle=	captionstyle=italic captionstyle=bold captionstyle="bold italic"	This option specifies the text style to be used in the caption. You can choose one or more options from bold, italic, underline, outline and shadow. All of the text in the caption will be displayed in the same style, you cannot mix styles.
captioncolor=	captioncolor=#00FF00 captioncolor=red captioncolor=royalblue	This option specifies the caption text color (the default is black). If the color begins with # then you can specify any color using HTML style color tags (#RRGGBB, for example #00FF00 for green). You can also choose from this list of colors: <i>aliceblue, antiquewhite, aqua, aquamarine, azure, beige, bisque, black, blanchedalmond, blue, blueviolet, brown, burlywood, cadetblue, chartreuse, chocolate, coral, cornflowerblue, cornsilk, crimson, cyan, darkblue, darkcyan, darkgoldenrod, darkgray, darkgreen, darkkhaki, darkmagenta, darkolivegreen, darkorange, darkorchid, darkred, darksalmon, darkseagreen, darkslateblue, darkslategray, darkturquoise, darkviolet, deeppink, deepskyblue, dimgray, dodgerblue, firebrick, floralwhite, forestgreen, fuchsia, gainsboro, ghostwhite, gold, goldenrod, gray, green, greenyellow, honeydew, hotpink, indianred, indigo, ivory, khaki, lavender, lavenderblush, lawngreen, lemonchiffon, lightblue, lightcoral, lightcyan, lightgoldenrodyellow, lightgreen, lightgrey, lightpink, lightsalmon, lightseagreen, lightskyblue, lightslategray, lightsteelblue, lightyellow, lime, limegreen, linen, magenta, maroon, mediumaquamarine, mediumblue, mediumorchid, mediumpurple, mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise, mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, navajowhite, navy, oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen, paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, powderblue, purple, red, rosybrown, royalblue, saddlebrown, salmon, sandybrown, seagreen, seashell, sienna, silver, skyblue, slateblue, slategray, snow, springgreen, steelblue, tan, teal, thistle, tomato, turquoise, violet, wheat, white, whitesmoke, yellow, yellowgreen.</i>
buttons=	buttons="Yes;No;Cancel" buttons="One:50;Two:50"	This option allows you to specify up to three buttons (the default is two buttons: <i>Ok</i> and <i>Cancel</i>). Each button is separated by a semicolon, and the first button listed is the default button. The default button width is 80 pixels, you can also specify a button width in pixels by placing a colon followed by the width after the button name. In the second example to the left the buttons will be 50 pixels wide.
bgcolor=	bgcolor=#FFDDFF bgcolor=lightskyblue bgcolor=##983	This option specifies the background color (the default is gray). If the color begins with # then you can specify any color using HTML style color tags (#RRGGBB, for example #FFCCFF for lite pink). You can also choose from the same list of colors available for the color= option (see above). A third option is to use ## followed by a resource number to display a background image from Panorama's resources. You can use the Icons & Backgrounds wizard to find out what background images are available.

After the dialog is complete the program can use the `info("dialogtrigger")` to find out which button was clicked.

Obsolete Text Entry Statements

Since Panorama was first released in 1988 it has evolved and improved in many ways. For compatibility, we usually leave in older features even when new features make them obsolete, as is the case with the statements described in this section. However, they are still described here because you may encounter them in older databases. The `getscrap` statement displays a simple dialog.



The user types in one item of text, then presses **OK** or **Stop**. Panorama will put whatever text the user types into the clipboard. The `getscrap` statement has one parameter, the text that you want to appear at the top of the dialog. The example below uses `getscrap` to find out what check to search for.

```
getscrap "Find what check #?"
select «Check#»=val(clipboard())
```

The `getscrapok` statement is similar to `getscrap`, but the dialog has no **Stop** button.

The SuperChoiceDialog Statement

This statement displays a configurable dialog that displays a list of choices. This statement has three parameters:

```
superchoicedialog list,selection,options
```

The **list** parameter must be a text delimited array containing the lists of choices that will be displayed. The **selection** parameter must be a field or variable to accept the choice. The **options** is a text parameter that uses a syntax similar to an HTML tag to specify one or more options. If you just want to use the default options you don't have to specify any options at all.

```
superchoicedialog States,theState,""
```

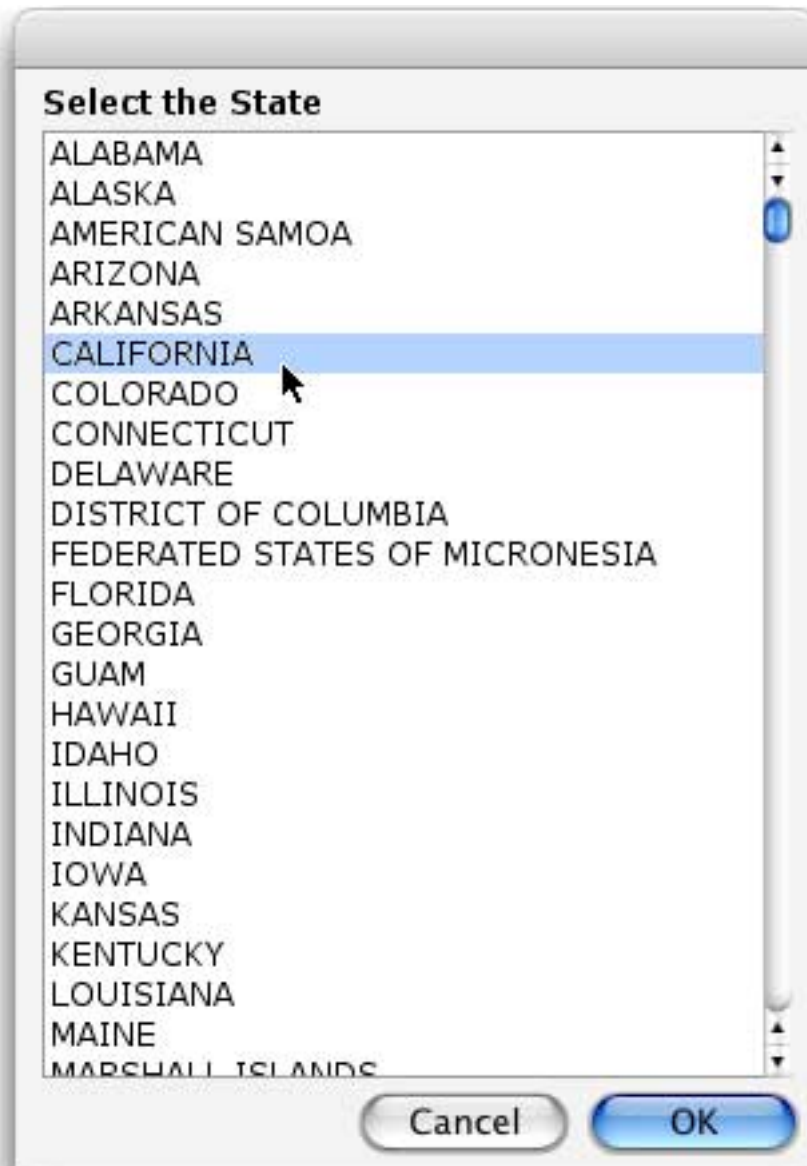
This plain dialog looks like this:



Each option is specified as an **option=value** pair, for example **height=4in**, **caption="Enter Quantity"**, etc. Here is an example of a dialog with several options set. (Tip: If you enclose the entire option parameter in "smart quotes" as shown below you will be able to use regular " quotes for the individual options, if necessary.)

```
superchoicedialog States,theState,  
  "Caption="Select the State" captionstyle=bold height=6in width=3in"
```

This dialog will look like this.



The table below describes each of options available with this statement.

Option	Examples	Description
title=	title="Latest Information"	With this option you can set the title of the dialog. This appears in the title bar at the top of the dialog window.
height=	height=400 height=5in height=4.5" height=10cm height=80% height=-50 height=-2cm	This option specifies the height of the dialog. The default is 200 pixels. If you supply just a number then the height is specified in pixels (72 pixels = 1 inch). If the number is followed by in or " then it is specified in inches. If the number is followed by cm it is specified in centimeters. If the number is followed by % it is specified as a percentage of the screen height, for example 50% is 1/2 of the screen height. If the height is negative then this value is the distance of the border between the top of the screen and the alert (also the bottom of the alert and the bottom of the screen). For example if your screen is 8 inches high then a height of -1in will make the alert 6 inches high (one inch at the top and one inch at the bottom).
width=	width=600 width=7in width=5.5" width=12cm width=80% width=-50 width=-2cm	This option specifies the width of the dialog. The default is 500 pixels. If you supply just a number then the width is specified in pixels (72 pixels = 1 inch). If the number is followed by in or " then it is specified in inches. If the number is followed by cm it is specified in centimeters. If the number is followed by % it is specified as a percentage of the screen width, for example 50% is 1/2 of the screen width. If the width is negative then this value is the distance of the border between the left of the screen and the alert (also the right edge of the alert and the right side of the screen). For example if your screen is 10 inches wide then a width of -1in will make the alert 8 inches wide (one inch on the left and one inch on the right).
font=	font=Tekton font="Comic Sans" font=Verdana	This option specifies the font to be used for the list of choices. The default is whatever the system font is on your system.
size=	size=9 size=18	This option specifies the text size (in points) for the list of choices. The default is 12 points.
caption=	caption="Start date:"	This option specifies a message to be displayed at the top of the dialog, above the text. (This message is displayed within the dialog, not in the title bar.)
captionheight=	captionheight=2	This option specifies how many lines should be used for the caption (see above). This value defaults to 1, but you can use a larger value if you have a long caption.

Option	Examples	Description
captionfont=	captionfont=Tekton captionfont="Comic Sans" captionfont=Verdana	This option specifies the font to be used in the caption. This allows you to specify a different font for the caption and the text editing portions of the dialog.
captionsize=	captionsize=9 captionsize=18	This option specifies the text size (in points) to be used in the caption. This allows you to specify a different font size for the caption and the text editing portions of the dialog.
captionstyle=	captionstyle=italic captionstyle=bold captionstyle="bold italic"	This option specifies the text style to be used in the caption. You can choose one or more options from bold, italic, underline, outline and shadow. All of the text in the caption will be displayed in the same style, you cannot mix styles.
captioncolor=	captioncolor=#00FF00 captioncolor=red captioncolor=royalblue	This option specifies the caption text color (the default is black). If the color begins with # then you can specify any color using HTML style color tags (#RRGGBB, for example #00FF00 for green). You can also choose from this list of colors: <i>aliceblue, antiquewhite, aqua, aquamarine, azure, beige, bisque, black, blanchedalmond, blue, blueviolet, brown, burlywood, cadetblue, chartreuse, chocolate, coral, cornflowerblue, cornsilk, crimson, cyan, darkblue, darkcyan, darkgoldenrod, darkgray, darkgreen, darkkhaki, darkmagenta, darkolivegreen, darkorange, darkorchid, darkred, darksalmon, darkseagreen, darkslateblue, darkslategray, darkturquoise, darkviolet, deeppink, deepskyblue, dimgray, dodgerblue, firebrick, floralwhite, forestgreen, fuchsia, gainsboro, ghostwhite, gold, goldenrod, gray, green, greenyellow, honeydew, hotpink, indianred, indigo, ivory, khaki, lavender, lavenderblush, lawngreen, lemonchiffon, lightblue, lightcoral, lightcyan, lightgoldenrodyellow, lightgreen, lightgrey, lightpink, lightsalmon, lightseagreen, lightskyblue, lightslategray, lightsteelblue, lightyellow, lime, limegreen, linen, magenta, maroon, mediumaquamarine, mediumblue, mediumorchid, mediumpurple, mediumseagreen, mediumslateblue, mediumspringgreen, mediumturquoise, mediumvioletred, midnightblue, mintcream, mistyrose, moccasin, navajowhite, navy, oldlace, olive, olivedrab, orange, orangered, orchid, palegoldenrod, palegreen, paleturquoise, palevioletred, papayawhip, peachpuff, peru, pink, plum, powderblue, purple, red, rosybrown, royalblue, saddlebrown, salmon, sandybrown, seagreen, seashell, sienna, silver, skyblue, slateblue, slategray, snow, springgreen, steelblue, tan, teal, thistle, tomato, turquoise, violet, wheat, white, whitesmoke, yellow, yellowgreen.</i>
buttons=	buttons="Yes;No;Cancel" buttons="One:50;Two:50"	This option allows you to specify up to three buttons (the default is two buttons: <i>Ok</i> and <i>Cancel</i>). Each button is separated by a semicolon, and the first button listed is the default button. The default button width is 80 pixels, you can also specify a button width in pixels by placing a colon followed by the width after the button name. In the second example to the left the buttons will be 50 pixels wide.
bgcolor=	bgcolor=#FFDDFF bgcolor=lightskyblue bgcolor=##983	This option specifies the background color (the default is gray). If the color begins with # then you can specify any color using HTML style color tags (#RRGGBB, for example #FFCCFF for lite pink). You can also choose from the same list of colors available for the color= option (see above). A third option is to use ## followed by a resource number to display a background image from Panorama's resources. You can use the Icons & Backgrounds wizard to find out what background images are available.

After the dialog is complete the program can use the `info("dialogtrigger")` to find out which button was clicked.

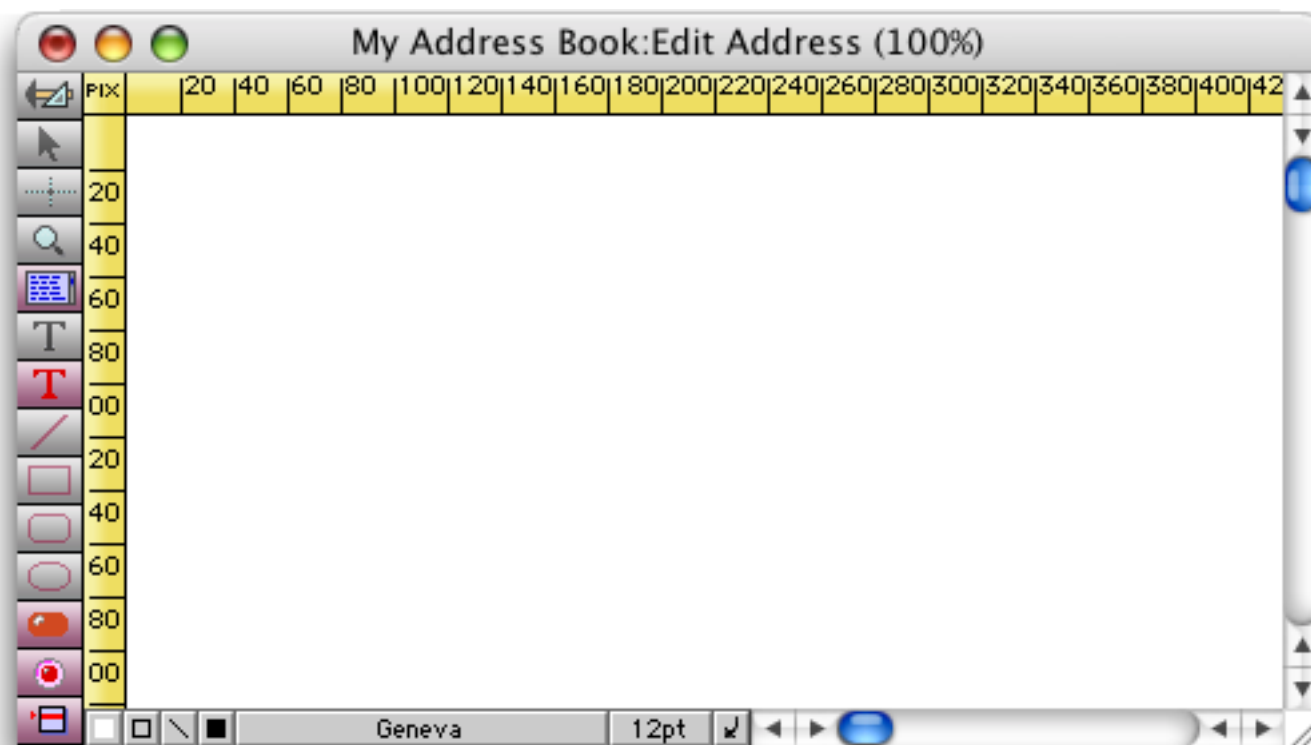
Custom Dialogs

Most of the dialogs you will need will not fit into the “off the shelf” category described in the last section. When an off the shelf dialog won't cut it, you can build your own dialogs using standard Panorama forms. Forms used as dialogs are created just like any other form, using text objects, buttons, lists, pop-up menus, pictures, etc. In fact, any form can be used as a dialog.

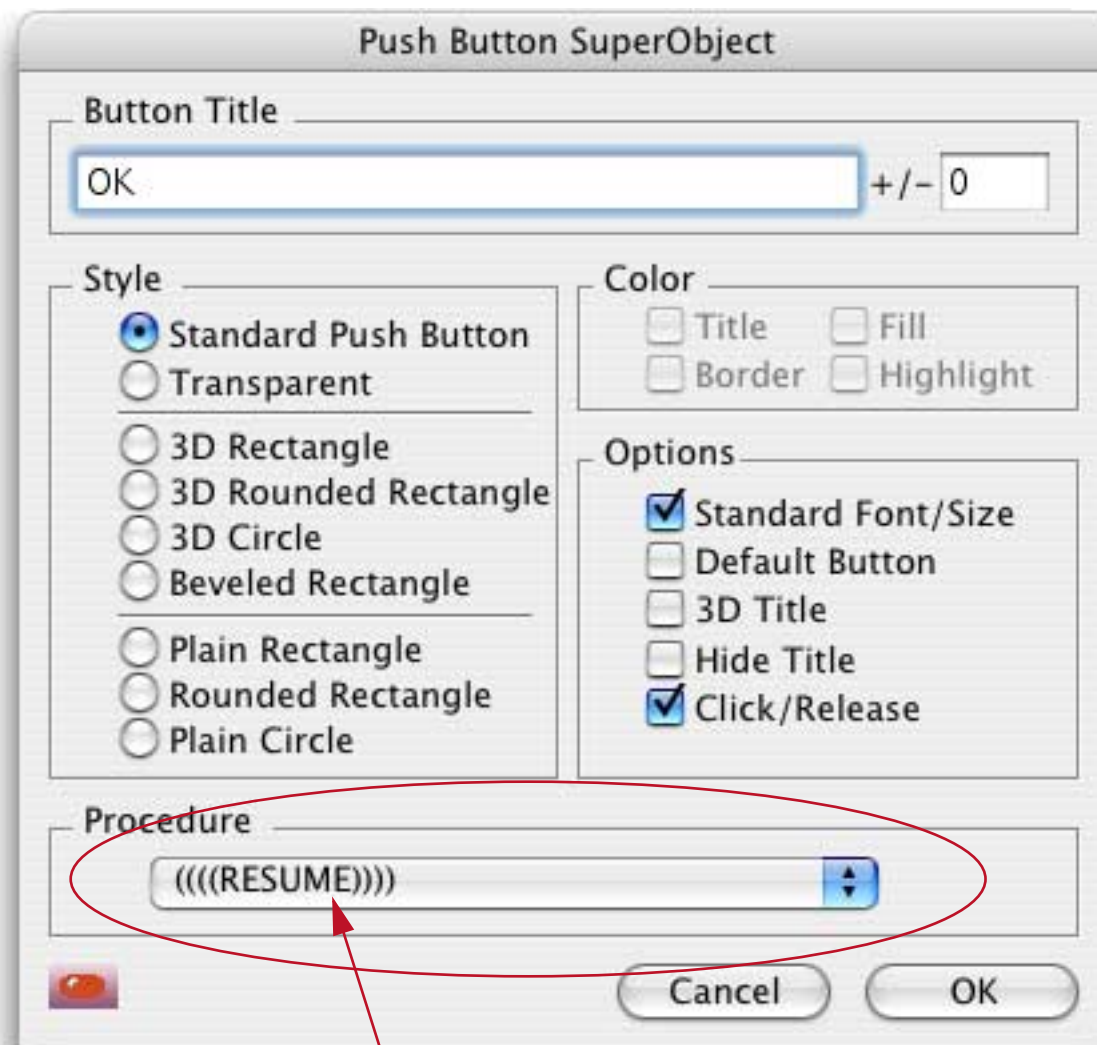
If a dialog is going to be used to collect information that is independent from the database (i.e. not in a database field) your dialog should use SuperObjects™ that are linked to global variables. You can use the Super-Object Text Editor, Pop-Up Menus, Data Buttons and Lists with global variables. When the dialog is closed, the procedure can use the information the user entered into these global variables any way it wants to.

Preparing a Form for Use as a Dialog

The first step in designing a custom dialog is to create a normal form (see “[Creating a New Form, Crosstab or Procedure](#)” on page 182 of the *Panorama Handbook*).

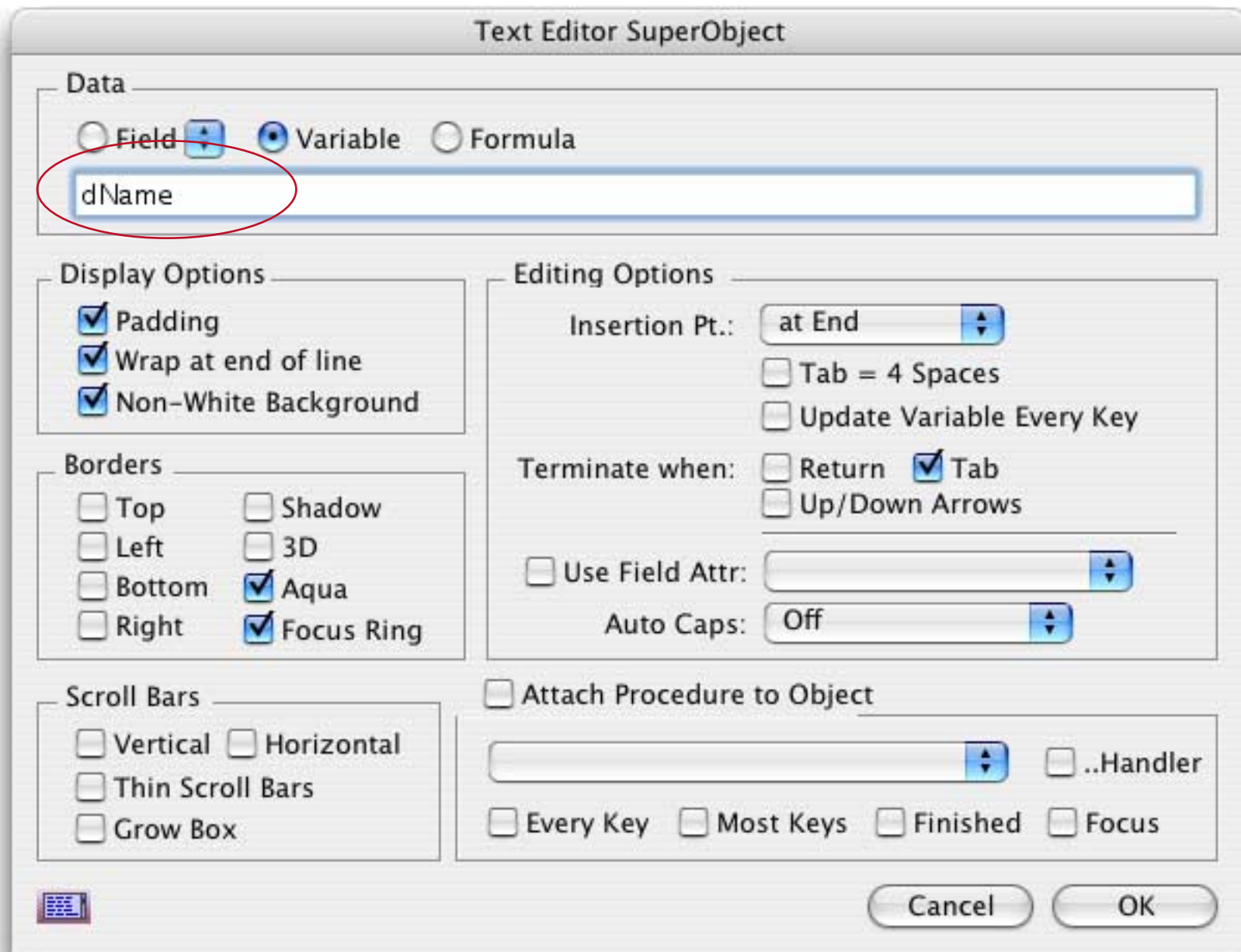


Next use the **Push Button** tool (see “[Super Object Push Button](#)” on page 823 of the *Panorama Handbook*) to create the **OK** and **Cancel** buttons. Both of these buttons should be configured to resume the current procedure when clicked. To set up this configuration on the Macintosh hold down the **Control** key and click on the **Procedure** pop-up menu. To setup this configuration on Windows simply right click on the **Procedure** pop-up menu.

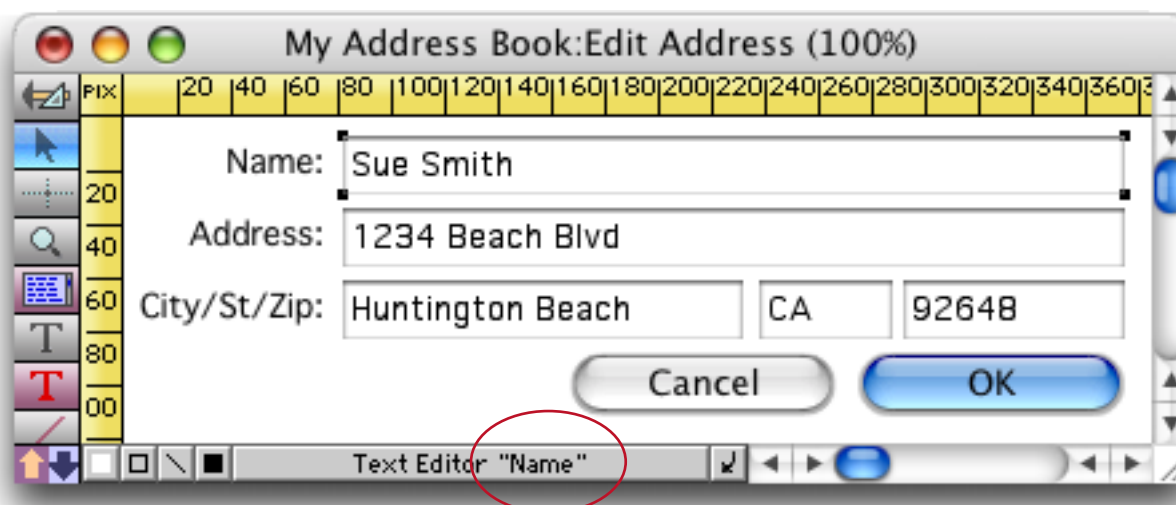


*MacOS — hold down Control key and click
Windows — right click*

The next step is to add any text editing boxes that are needed using the Text Editor SuperObject (see “[Text Editor SuperObject](#)” on page 639 of the *Panorama Handbook*). If a text editing box is going to be used to edit a database field it should be configured to edit a variable. The variable should have the same name as the field but with a prefix of **d**. The example below shows the configuration for editing the **Name** field.

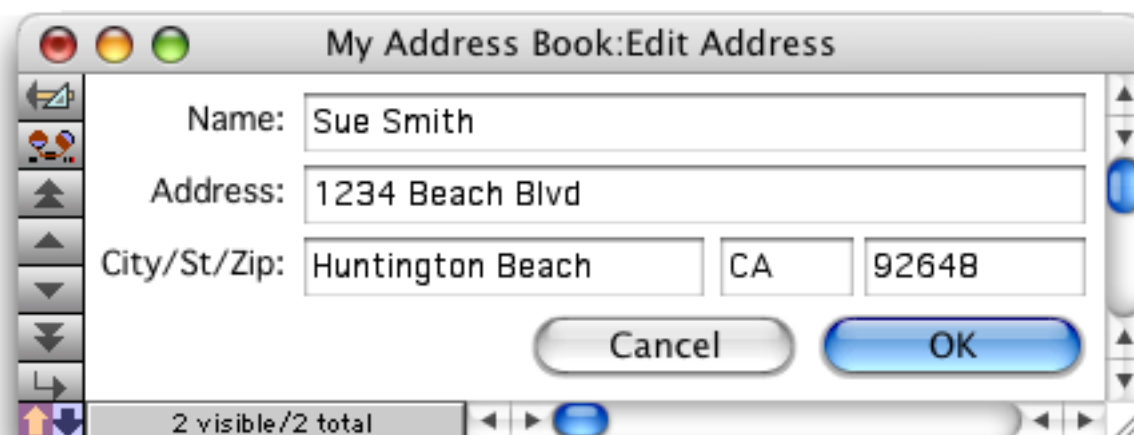


An optional step at this point is to give some or all of the text editor objects a name (see “[Object Type/Object Name](#)” on page 533 of the *Panorama Handbook* to learn how to assign a name to any object). In particular you’ll probably want to assign a name to the upper left object, the object that will become the default for text entry when the dialog is first opened.

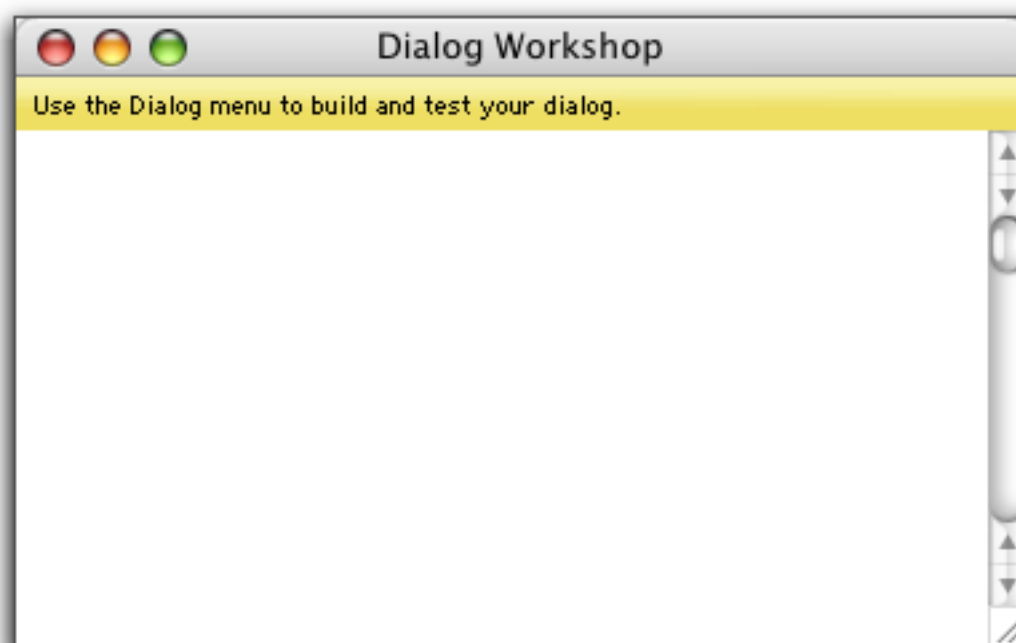


At this point you can add text captions and any checkboxes or radio buttons. To configure a checkbox or radio button to edit a database field it should be a variable with the same name as the field but with a **d** prefix, just as for the Text Editor SuperObjects. Otherwise the variable can be any name you like.

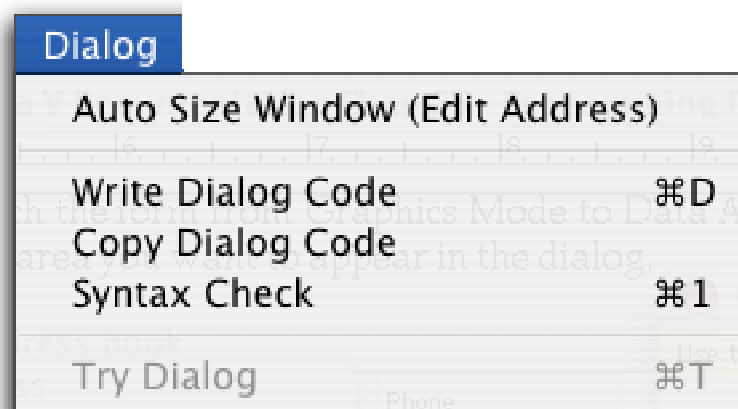
Once all the objects on the form are complete, switch the form from Graphics Mode to Data Access Mode. Then adjust the size of the window to show just the area you want to appear in the dialog.



Now open the **Dialog Workshop** (in the Developer Tools subfolder of the Wizard menu). This opens a small window.

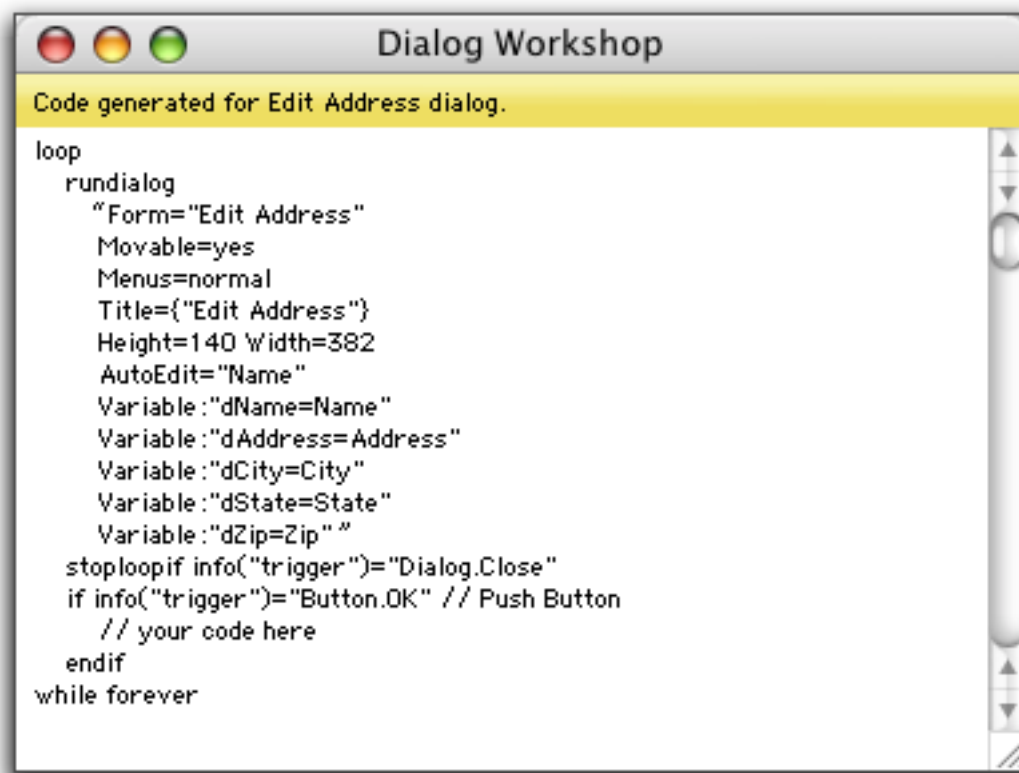


The guts of this workshop is in the **Dialog** menu.



If you haven't already set the size of your window you can use the **Auto Size Window** command to adjust the form size. The wizard will try its best to adjust the form to an appropriate size.

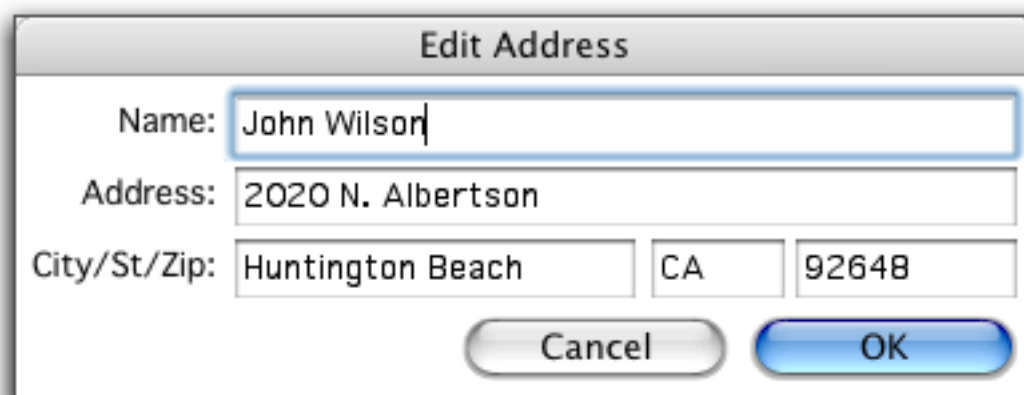
When your form is ready to go choose the **Write Dialog Code** command. This command will analyze the form and write the code for a new procedure for you!



```
Code generated for Edit Address dialog.

loop
  rundialog
    "Form="Edit Address"
    Movable=yes
    Menus=normal
    Title={"Edit Address"}
    Height=140 Width=382
    AutoEdit="Name"
    Variable:"dName=Name"
    Variable:"dAddress=Address"
    Variable:"dCity=City"
    Variable:"dState=State"
    Variable:"dZip=Zip"
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger")="Button.OK" // Push Button
    // your code here
  endif
while forever
```

To try out your new dialog use the **Try Dialog** command. This allows you to see exactly what your new dialog looks like and how it will work.



Edit Address			
Name:	John Wilson		
Address:	2020 N. Albertson		
City/St/Zip:	Huntington Beach	CA	92648
Cancel		OK	

Next, close the form window. However, make sure that you leave another window in the database open, so that the database itself doesn't become closed.

If your dialog works the way you expected the next step is to copy the procedure into the database itself. You can copy your new procedure to the clipboard with the **Copy Dialog Code** command. If the dialog will be opened by clicking on a button or selecting an item in the **Action** menu then you'll need to create a new procedure (see "[Writing a Procedure from Scratch](#)" on page 216). Once the procedure is opened you can use the **Paste** command to insert the automatically generated code into it.

```

loop
  rundialog
    "Form="Edit Address"
    Movable=yes
    Menus=normal
    Title={"Edit Address"}
    Height=140 Width=382
    AutoEdit="Name"
    Variable:"dName=Name"
    Variable:"dAddress=Address"
    Variable:"dCity=City"
    Variable:"dState=State"
    Variable:"dZip=Zip"
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger")="Button.OK" // Push Button
    // your code here
  endif
while forever

```

For most dialogs, that's it! Once you've tried one or two dialogs you'll find that you can create a new dialog in just a few minutes.

Before you actually use the dialog you'll need to close the form you created. Make sure that some other window in this database remains open. To use the dialog you can simply select the new procedure from the **Action** menu (or click on the button or whatever). The dialog will appear, and will automatically be centered over the current window (or, if the window is too small, centered in the middle of the screen). Any cells or buttons associated with a field will automatically be filled in with the original data from the current record.

Edit Address		
Name:	John Wilson	
Address:	2020 N. Albertson	
City/St/Zip:	Huntington Beach	CA 92648
Cancel		OK

When you press the **OK** or **Cancel** buttons the dialog will close automatically, and if the **OK** button was pressed any fields that were modified will be updated.

Customizing the Dialog Code

The program automatically generated by the **Dialog Workshop** will handle many common dialogs as-is. However, this program is designed to be flexible and to allow you to modify almost all aspects of its behavior. The key to this procedure is the **RunDialog** statement. Let's start by looking at the automatically generated code to see how it works. Here is the simplest possible dialog processing code — just four lines.

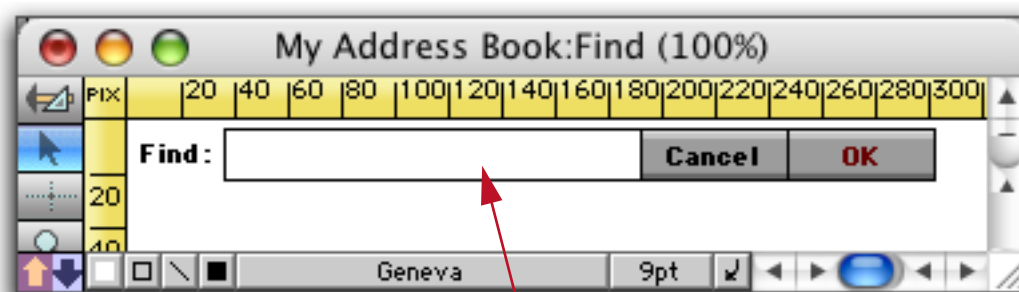
```
loop
  rundialog, {Form=Address Height=120 Width=400 Movable=yes Menus=normal}
  stoploopif info("trigger")="Dialog.Close"
while forever
```

In this most basic form the code is a simple loop that executes the **rundialog** statement each time through the loop. This statement handles most of the work. Whenever something happens (a button is pressed, the **Enter** or **Tab** key is pressed), the **rundialog** statement analyzes it and then returns to the loop. Your code in the loop can find out what is happening by examining the result of the **info("trigger")** function.

The **rundialog** statement has one parameter — a list of options. The **options** is a text parameter that uses a syntax similar to an HTML tag to specify one or more options. Each option is specified as an **option=value** pair, for example **form="Entry"**, **title="Enter Quantity"**, etc. (Tip: If you enclose the entire option parameter in "smart quotes" you will be able to use regular " quotes for the individual options, if necessary.)

This very simple program shown above only does one thing — stop the loop if **info("trigger")** becomes **Dialog.Close**. Our code doesn't need to do anything else because the **rundialog** statement will do everything for us including opening and closing the dialog.

To illustrate how this procedure can be expanded, let's consider the dialog show below. This dialog is designed to allow the user to type in a word or phrase they want to search for.



linked to findThis global variable

Here's the procedure that can handle this dialog. The automatically generated code is shown in blue, the custom code in purple.

```
global findThis
findThis=""
loop
  rundialog {Form="Find" Height=45 Width=346 movable=yes menus=normal AutoEdit="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    find exportline() contains findThis
  endif
while forever
```

The first two lines simply create the global variable named **findThis** and assign it a value. This is the variable the user will type into. The other new code checks to see if the **OK** button has been pressed, and if so, performs the search.

We can modify this code further to perform error checking. This version of the program checks to make sure that the user has typed something to search for. If not a message is displayed. More importantly, the `settrigger` statement is set to "" instead of `Dialog.OK`. This tells the `rundialog` statement not to close the dialog window.

```
global findThis
findThis=""
loop
  rundialog {Form="Find" Height=45 Width=346 movable=yes menus=normal AutoEdit="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    if findThis=""
      message "You must enter something to search for!"
      settrigger ""
    else
      find exportline() contains findThis
    endif
  endif
endif
while forever
```

The way the code above is written the find operation happens while the dialog is still open. If you wanted the dialog to close first you would need to rewrite the program like this.

```
global findThis
findThis=""
loop
  rundialog {Form="Find" Height=45 Width=346 movable=yes menus=normal AutoEdit="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    if findThis=""
      message "You must enter something to search for!"
      settrigger ""
    endif
  endif
endif
while forever
  if dlgResult="Ok"
    find exportline() contains findThis
  endif
endif
```

When the dialog is finished the `dlgResult` value will contain either the value `Ok` or `Cancel`.

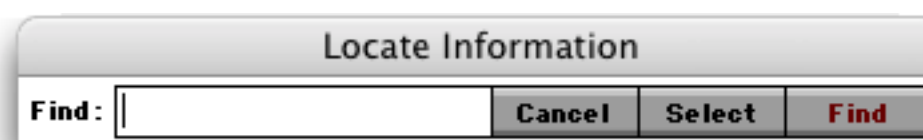
You can add additional buttons to the dialog that perform some action within the dialog. For example, you could add a **Clear** button to this dialog.



When a button other than the **OK** or **Cancel** button is pressed the `info("trigger")` function will return **Button**, followed by the title of the button.

```
global findThis
findThis=""
loop
  rundialog {Form="Find" Height=45 Width=346 movable=yes menus=normal AutoEdit="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    if findThis=""
      message "You must enter something to search for!"
      settrigger ""
    else
      find exportline() contains findThis
    endif
  endif
  if info("trigger") = "Button.Clear"
    superobject "Find","Open"
    activesuperobject "setselection",0,32767
    activesuperobject "clear"
  endif
while forever
```

Sometimes you may want to have more than one button that terminates the dialog successfully. In this case both the **Find** and **Select** buttons cause the dialog to close.



Here is the revised program to handle this dialog.

```
global findThis
findThis=""
loop
  rundialog
    {Form="Find" Height=45 Width=346 movable=yes menus=normal
      AutoEdit="Find" OkButton="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    if findThis=""
      message "You must enter something to search for!"
      settrigger ""
    else
      find exportline() contains findThis
    endif
  endif
  if info("trigger") contains "Button.Select"
    superobjectclose
    if findThis=""
      message "You must enter something to select!"
    else
      select exportline() contains findThis
      settrigger "Dialog.OK"
    endif
  endif
while forever
```

The first thing to notice is the option `OkButton="Find"` on the fourth line. Options are discussed in more detail in the next section, but for now this option tells the `rundialog` statement to treat the `Find` button as if it was the `OK` button. That means that when you press the `Enter` or `Return` key it will be treated just as if you had pressed the `Find` button. It also means that when the `Find` button is pressed the `info("trigger")` function will return `Dialog.OK`, not `Button.Find` (see line 6).

The additions to handle the `Select` button are fairly routine. However, notice the fourth line from the bottom, `settrigger "Dialog.OK"`. This line tells the `rundialog` statement to go ahead and close the dialog window.

In some cases you may need to perform some initialization after the dialog window has opened. Usually this involves some sort of graphic manipulation — moving an object or changing a font (see “[Programming Graphic Objects on the Fly](#)” on page 633). (Almost any other kind of non-graphic initialization can simply be performed before the loop begins.) We don’t have an example of this, but the basic idea is to check for the trigger value of `Dialog.Initialize`.

```
loop
  rundialog {Form="Find" Height=45 Width=346 movable=yes menus=normal AutoEdit="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.Initialize"
    /*
     ... code to initialize procedure goes here ...
    */
  endif
while forever
```

Sometimes you may want to handle the `Cancel` button in a special way. The revised procedure below checks to see if the user has typed anything in, and if so, asks them to confirm that they really do want to cancel.

```
global findThis
findThis=""
loop
  rundialog {Form="Find" Height=45 Width=346 movable=yes menus=normal
    AutoEdit="Find" OkButton="Find"}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    if findThis=""
      message "You must enter something to search for!"
      settrigger ""
    else
      find exportline() contains findThis
    endif
  endif
  if info("trigger") contains "Button.Select"
    superobjectclose
    if findThis=""
      message "You must enter something to select!"
    else
      select exportline() contains findThis
      settrigger "Dialog.OK"
    endif
  endif
  if info("trigger") = "Dialog.Cancel"
    superobjectclose
    if findThis=""
      alert 1014,"Are you sure you want to cancel?"
      if info("dialogtrigger") contains "no"
        settrigger "" /* tell .dialog to stop the cancel! */
        superobject "Find","Open"
      endif
    endif
  endif
while forever
```

All of the examples have shown push buttons, but you can also check for and handle any type of button, list, or even a Text Editor SuperObject that triggers the `rundialog` statement using `((((Resume))))`. Any object that has the `((((Resume))))` option set can be handled by your custom dialog code.

Options to the RunDialog Statement

The `rundialog` statement has one parameter. This parameter contains a series of `name=value` pairs that tell the `.dialog` subroutine how to process the dialog. At a minimum this parameter must include three parameters: the form name, the form height, and the form width.

```
{Form=Address Height=120 Width=400}
```

If the form name (or any value) contains a space it must be surrounded with quotes, like this.

```
{Form="Time Card" Height=120 Width=400}
```

In addition to the three basic name/value pairs there are also about a dozen other optional name/value pairs that you can specify to customize the appearance and behavior of your dialog.

The `height` and `width` options are normally specified in pixels. (One inch is equal to 72 pixels). However, you can also specify these values in inches (for example `2"` or `3in`), centimeters (`12cm`) or as a percentage of the screen width or height (`75%`). If the value is negative then the dialog size will depend on the height and/or width of the screen. For example, `height=-72` specifies a dialog with 1 inch above and below the dialog. If the dialog has a variable size (either a percentage or negative value) then you should make the form elastic (see “[Elastic Forms](#)” on page 922 of the *Panorama Handbook*) so that it can adjust to whatever screen size is available.

The `movable` option allows you to create a dialog with a drag bar that can be moved around on the screen. The value for this option should be yes or no, for example

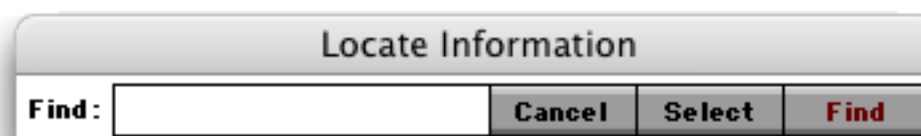
```
movable=yes
```

The default is yes, and we no longer recommend turning this option off.

If you don't give the dialog a title it will use the name of the form, as shown above. You can override this and specify another title using the `windowtitle` option, like this.

```
windowtitle="Locate Information"
```

The dialog will appear with the specified name in the title bar.



When a dialog has more than one editable text item normally the top left item is the default item where you will begin typing.

If the Text Editor SuperObjects are named (see “[Object Type/Object Name](#)” on page 533 of the *Panorama Handbook*) you can override this default and specify a different default editing item with the `autoedit` option.

```
autoedit=Zip
```

With this option set the [Zip Code](#) becomes the default item.

The `autoeditstart` and `autoeditend` options control what text is initially selected in the default item.

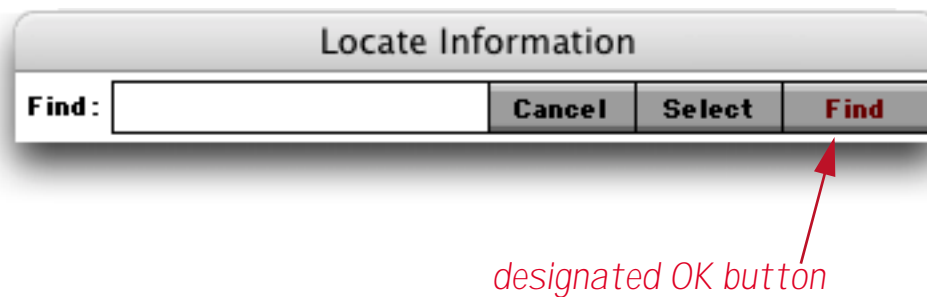
```
autoeditstart=0 autoeditend=0
```

If both of these values are set to zero the initial editing point will be at the beginning of the text. If both of these are set to a large value like 9999 the initial editing point will be at the end of the text.

The `okbutton` option allows you to change what button is considered the **OK** button.

```
okbutton=Find
```

The **OK** button is usually named **OK** but it can be changed to any button on the form. When the button designated as the **OK** button is pressed the `info("trigger")` function will return `Dialog.OK`, even if the actual button has a different name. In addition, pressing the **Enter** or **Return** key will be treated the same as clicking on whatever button has been designated as the **OK** button.



The `cancelbutton` option allows you to change what button is considered the **Cancel** button.

```
cancelbutton=Stop
```

The **Cancel** button is usually named **Cancel** but it can be changed to any button on the form. When the button designated as the **Cancel** button is pressed the `info("trigger")` function will return `Dialog.Cancel` even if the actual button has a different name.

The `timeout` option allows you to put a time limit on a dialog. The timeout value is specified in seconds, so this option will cause the dialog to time out in two minutes.

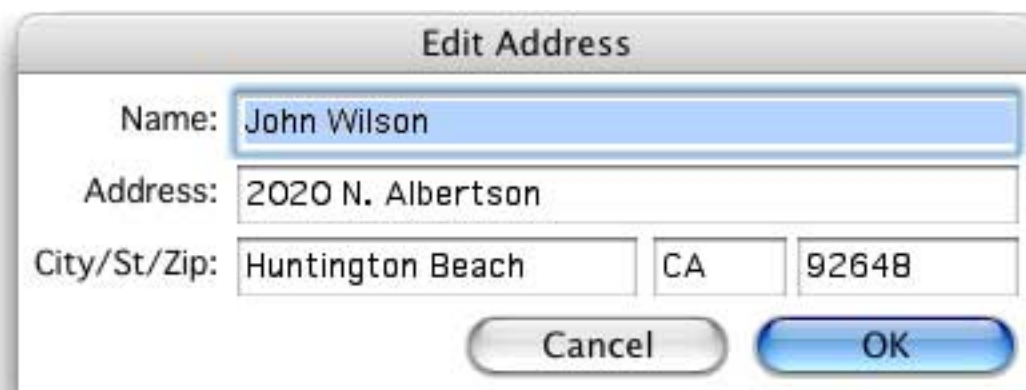
```
timeout=120
```

If the dialog is still open after two minutes it will close automatically as if the designated Ok button was pressed.

Editing Database Information with a Dialog

Editing database information with a dialog that has **OK** and **Cancel** buttons takes some extra effort. You can't simply edit the data directly because if the user presses the **Cancel** button you must be able to restore the original data. The solution is to copy the data from the database into variables, edit the variables, and then only copy the data back into the database if the **OK** button is pressed. You can write code for all this yourself, but it's easier to let the `rundialog` statement take care of it all for you.

To illustrate this, consider this dialog for editing an address.



The dialog edits five database fields — **Name**, **Address**, **City**, **State** and **Zip**. If you followed instructions carefully you have set up the SuperObject Text Editors in this dialog to edit five corresponding variables — **dName**, **dAddress**, **dCity**, **dState** and **dZip** (see “[Preparing a Form for Use as a Dialog](#)” on page 489). Now one way to set this up would be to write the code to transfer the data back and forth yourself.

```
global dName,dAddress,dCity,dState,dZip
dName=Name
dAddress=Address
dCity=City
dState=State
dZip=Zip
loop
  rundialog {Form=Address Height=120 Width=400 menus=normal AutoEdit=Name}
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") contains "Dialog.OK"
    Name=dName
    Address=dAddress
    City=dCity
    State=dState
    Zip=dZip
  endif
while forever
```

This is a lot of extra work, though, because you have to type the field names twice and the variable names three times! Another option is to declare the relationship between the fields and variables as part of the option parameter to the **rundialog** statement. Each declaration takes the form

```
Variable:"<variable>=<field>"
```

Here is our revised procedure. A lot shorter, eh? Make sure that these declarations are in the parameter to the **rundialog** statement, between the { and } characters. On the other hand, if you set up your form correctly the **Dialog Workshop** will write all of the declarations for you, completely automatically!

```
loop
  rundialog {Form=Address Height=120 Width=400 AutoEdit=Name
    Variable:"dName=Name"
    Variable:"dAddress=Address"
    Variable:"dCity=City"
    Variable:"dState=State"
    Variable:"dZip=Zip"}
  stoploopif info("trigger")="Dialog.Close"
while forever
```

Sometimes the data needs to be converted in addition to being copied. Any time a dialog needs to edit a numeric or date field the declaration needs to include the functions for converting in both directions. Here's how a numeric **Amount** field and date **StartDate** field would be handled.

```
Variable:"val(«dAmount»)=str(«Amount»)"
Variable:"date(«dStartDate»)=datepattern(«StartDate», "mm/dd/yy")"
```

When conversion functions are used the variable and field names must always be enclosed in « and » chevrons (see “[Special Characters](#)” on page 57). The chevrons must be included even if the variable or field name doesn't contain any blanks or punctuation. If the chevrons are omitted an error will occur when you try to open the dialog.

The **rundialog** statement normally creates the variables you specify as global variables when the dialog is opened (see “[Long Life Variables](#)” on page 249). Using the **variabletype** option you can specify that another type of variable be created instead. The only option that makes any sense here is **fileglobal**.

```
variabletype=fileglobal
```

Custom Dialog Menus

The `rundialog` statement normally displays only the **Apple** (Mac only) and **Edit** menus when a dialog is open. This allows you to cut, copy and paste text within the menu. If you want to turn off all the menus, add `menu=none` to the list of options.

You can also specify your own completely custom menus using Panorama's Live Menu feature (see "[Live Menus](#)" on page 362). Use the `menu=` option followed by the formula you want to use, like this:

```
rundialog " ... other options ... menu={formula} ... other options ... "
```

In designing the formula you'll need to use the same rules as the for the second parameter of the `filemenubar` and `windowmenubar` statements (see "[The FileMenuBar Statement](#)" on page 362). Specifically, you'll usually want to use the `menu()`, `menuitems()`, `arraymenu()` and other similar functions to assemble the custom menu items. The example below adds a custom **City** menu when the dialog is open. This menu will list all of the cities in the database.

```
rundialog "Form="Find/Select" Height=45 Width=346 AutoEdit="Find" OkButton="Find"
  menus={menu("City")+arraymenu(listchoices(City,¶))}
  windowtitle="Locate Information"
```

Here is the dialog with the **City** menu pulled down. (Notice that the `rundialog` statement automatically includes the **Edit** menu.)

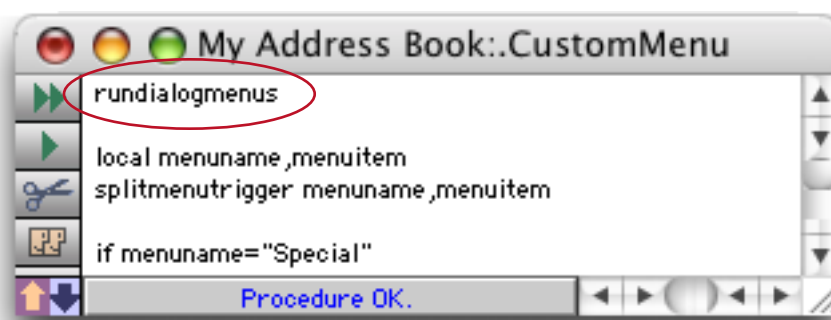


To make this custom menu work you'll need to add code to the dialog loop to check for the menu. This code is identical to the code used for the `.CustomMenu` procedure (see "[.CustomMenu](#)" on page 381). The purple code below shows a typical way to handle the menu.

```
global findThis
findThis=""
loop
  rundialog "Form="Find/Select" Height=45 Width=346 AutoEdit="Find" OkButton="Find"
    menus={menu("City")+arraymenu(listchoices(City,¶))}
    windowtitle="Locate Information"
  stoploopif info("trigger")="Dialog.Close"
  if info("trigger") beginswith "menu."
    local menuname,menuitem
    splitmenutrigger menuname,menuitem
    if menuname="City"
      superobject "Find","Open"
      activesuperobject "inserttext",menuitem
    endif
  endif
endloop

... rest of dialog event loop ...
```

You'll also need to make one change to the **.CustomMenu** code for this database. At the beginning of this procedure add one line — **rundialogmenus**.



Now the custom menu code is complete. When you open this dialog, you can use the **City** menu to type the name of a city into the dialog.

Accessing and Modifying the Database Structure (Fields)

Usually the database field structure is set up in advance, and procedures simply work with the fields as they have been defined. However, for single user databases it is possible for a procedure to add new fields, delete fields, or change the properties of existing fields. For example you may want to temporarily add a field to perform a calculation, then remove the field when the procedure is finished.

There are two techniques for modifying field structure in a procedure. The procedure can modify the structure directly using special statements, or it can open and modify the design sheet.

Getting Information About Field Structure

Before you actually modify the field structure you might want to know something about it. There are several functions that a procedure can use to learn about the structure of a database.

The `dbinfo()` function can gather a variety of information about any open database. This function has two parameters:

```
dbinfo(option,database)
```

The second parameter is the name of the database that you want information about. If the parameter is empty ("") the current database is assumed. This database must be currently open. If the database is not open the `dbinfo()` function will not return any information.

The first parameter specifies the type of data you are requesting. Information types you may request include fields, forms, procedures, crosstabs, flash art, and folder. When the option is "fields" the function will return a carriage return separated list of the fields in the database (see "Text Arrays" on page 93). For example, the procedure below will display the number of fields in the current database:

```
message "The database "+info("databasename")+ " contains "+
      str(arraysize(dbinfo("fields",""),¶))+ " fields."
```

The `datatype()` function returns the data type of a field or variable—text, numeric, date, etc. This function has one parameter: the name of the field or variable in question. Depending on the data type this function will return one of the 10 values listed below.

Text	Integer
Choice	Fixed 1 Digit (##)
Picture	Fixed 2 Digits (###)
Date	Fixed 3 Digits (####)
Floating Point	Fixed 4 Digits (#####)

The procedure listed below uses the `datatype()` function and the `dbinfo()` function to build a list of all the numeric fields in the current database.

```
local X,XField,XType,AllFields,NumericFields
X=1 NumericFields=""
AllFields=dbinfo("fields","")
loop
  XField=array(AllFields,X,¶)
  stoploopif XField=""
  XType=datatype(XField)
  if XType beginswith "F" or XType beginswith "I"
    NumericFields=sandwich(" ",NumericFields,",")+XField
  endif
  X=X+1
while forever
message "Numeric Fields: "+NumericFields
```

Notice that `XField` is surrounded by an extra pair of parentheses when it is used in the `datatype()` function. Without this extra pair the `datatype()` function would return the type of the `XField` variable itself, instead of the field whose name is contained inside of `XField`.

The `info("fieldname")` function returns the name of the currently active field. You can use this function to save the current field name in a variable, go somewhere else, then return to the original spot.

Modifying Field Structure Directly

There are five statements that allow a procedure to modify the structure of a database directly: `addfield`, `insertfield`, `deletefield`, `fieldname`, and `fieldtype`.

The `addfield` statement adds one field to the end of the database (the extreme right edge of the data sheet). This statement has one parameter—the name of the new field:

```
addfield fieldname
```

The `fieldname` can be defined with any formula. To simply define a fixed field name, enclose that name in quotes like this:

```
addfield "Tax Rate"
```

The new field is always a text field. You can change it to a different data type with the `fieldtype` statement (see below). To make this new tax rate field a numeric two digit field the procedure would be modified like this.

```
addfield "Tax Rate"
field "Tax Rate"
fieldtype "Fixed 2 Digits (#.##)"
```

Note that the new field does not become the current field. Use the `field` statement to make the new field current before changing the type or performing other operations on the field, as shown in the example above.

The `insertfield` statement is exactly the same as the `addfield` statement except that the new field is inserted in front of the current field instead of being added to the end of the data sheet.

Both the `addfield` statement and the `insertfield` statement can be programmed to display a dialog allowing the user to set up the field name, type, etc. To use this option put the word `dialog` (no quotes) after the statement, like this:

```
insertfield dialog
```

To change the data type of the currently active field use the `fieldtype` statement. This statement has one parameter, the new data type:

```
fieldtype type
```

The `type` parameter is actually a text item that names the parameter. Legal field types are shown in this table.

Text	Integer
Choice	Fixed 1 Digit (##)
Picture	Fixed 2 Digits (###)
Date	Fixed 3 Digits (####)
Floating Point	Fixed 4 Digits (#####)

If the current field has any data in it, Panorama will attempt to convert the data to the new data type. If some of the data can't be represented in the new data type, that data will be thrown away; so be careful! For example if a text field is converted to date or number, data values like **John Smith** or **San Francisco** in that field will be tossed. Don't change the field type unless you are sure the data currently in the field can be converted, or unless you don't care!

The procedure below shows how these statements and functions can be used together. This procedure makes an exact copy of the current field. First it copies the current field name and type into the local variables **theField** and **theType** (the **datatype()** function is described in the previous section). Then it attempts to move one field to the right. If it can't (because the current field is the last field of the database) it adds a new field, otherwise it inserts a new field in the middle of the database. Finally, it sets the new field to the same data type as the original field and copies the data from the original field into the new field.

```
local theField,theType
theField=info("fieldname")
theType=datatype(info("fieldname"))
right
if stopped /* could also use if info("stopped") */
    addfield "Copy of "+theField
else
    insertfield "Copy of "+theField
endif
field ("Copy of "+theField)
fieldtype theType
formulafill grabdata("",theField)
```

To change the name of the current field use the **fieldname** statement (see "**FIELDNAME**" on page 5220). You can either specify the new name for the field using a formula, or use the word **dialog** to allow the user to enter the name in a dialog (they will also be able to modify other field properties.) (Note: Panorama will not prevent you from creating two or more fields with the same name. However, you should avoid this if possible. You can use the **dbinfo()** function to get a current list of the field names; see the previous section.)

To delete the current field, use the **deletefield** statement. Panorama won't display any warning—it will simply delete the field and all the data in it. Be careful because once you delete a field it's gone...you can't get it back. (Exception: If you have saved the database you might be able to get the field back with the **Revert To Saved** command.)

Hiding and Showing Fields

Panorama has several statements and functions for hiding and showing fields in the data sheet.

HideCurrentField - This statement hides the current field.

ShowAllFields - This statement shows all fields.

ShowTheseFields fieldlist - This statement shows all fields listed, leaving any other fields hidden. There should be one field name per line in the list. If none of your field names contain commas the **commatocr()** function is handy for this, for example:

```
showthesefields commatocr("Name,Phone,Email")
```

HideTheseFields fieldlist - This statement hides all fields listed, leaving any other fields visible. There should be one field name per line in the list. If none of your field names contain commas the **commatocr()** function is handy for this, for example:

```
hidethesefields commatocr("Salary,Birthday")
```

hiddenfields() - This function returns a carriage return delimited list of fields that are currently hidden in the current database, if any.

visiblefields() - This function returns a carriage return delimited list of fields that are currently hidden in the current database, if any.

Working With the Design Sheet

For the ultimate control the procedure can open the design sheet and change it just like any other database. There are two statements a procedure can use to open the design sheet: `opendesignsheet` and `godesignsheet`. The `opendesignsheet` statement opens the design sheet in a new window (see “[Opening a Window](#)” on page 445 for more information on opening windows). The `godesignsheet` statement opens the design sheet in the current window.

Once the design sheet is open, the procedure can locate any field it wants using the `find` statement (see “[Finding Information](#)” on page 552). Once the correct line is selected the procedure can change elements with assignment statements (`Default="Acme"`, etc.). When the changes are complete, the procedure must use the `newgeneration` statement to actually change the structure of the database.

The example below opens the design sheet and changes the output pattern for the `Price` field.

```
opendesignsheet
find <Field Name>="Price"
if info("found")
    <Output Pattern>="$#, .##"
    newgeneration
endif
closewindow
```

If you don't want the user to be able to see the shenanigans with the design sheet, use the `setwindow` or `setwindowrectangle` statements to make the window open outside the visible screen area (see “[Specifying the New Window Location](#)” on page 446).

Updating Database Structure From Another Database

Panorama includes a mechanism that lets you copy the structure of a database from another database while retaining the original data. Let's say that you have created a database and distributed it to many users far and wide...perhaps you are even selling the database. Your many users are each filling their databases with their own data. In the meantime, you are creating a new version. This new version of the database may have new fields, new forms, new procedures, and there are probably changes to existing forms/procedures/fields as well. Once you have finished your update you need a way to distribute the update and let each user update

his or her copy of the database so that it has the new structure but retains the old data. The **changenname**, **detachname** and **hijack** statements make this possible. The example procedure below shows how to do it. This procedure assumes the old version of the database is currently open. The procedure allows the user to locate the update file, then updates the structure.

```

local oldFile,newFolder,newFile,newFType
/* let user locate the update file */
openfiledialog newFolder,newFile,newFType,"KASXZEPD"
if newFile="" stop endif /* user pressed cancel */
/* save name of original database*/
oldFile=info("databasename")
/* change name of original database IN MEMORY ONLY */
changenname oldFile+".old"
/* open the file with the new structure */
openfile folderpath(newFolder)+newFile
/* suck the data from the old file into the new structure */
openfile "&"+oldFile+".old"
/* name of new database=name of old database (IN MEMORY ONLY) */
detachname oldFile
/* now connect to the original databases file on disk */
/* it's a "filejacking"! */
hijack oldFile+".old"
/* save the new, update file */
save
/* finally close the old database - we're done with it */
window oldFile+".old:SECRET"
closefile /* this file is really gone now */

```

If you look closely at this example, you will see that it doesn't really update the structure of the original database. Instead, it loads the data from the old database into the new database using Panorama's standard "append with matching names" feature (see "[Replacing the Data in a Database](#)" on page 407). Once this is done the old database is "detached" from its disk file. The new file then takes over or "hijacks" the detached disk file. As part of this "hijack" process Panorama also copies the auto-increment value, so if the database uses auto-numbering the numbers will continue to be generated in sequence.

Transferring Permanent Variables

If the original database has permanent variables that you want to keep, insert the following statements just before the **detachname** statement in the procedure above. This procedure assumes that the new updated database has at least the same permanent variables as the original database, and it copies the values from the old database to the new.

```

local oldPermanentVariables,opv,pv
/* build a list of the permanent variables */
oldPermanentVariables=dbinfo("permanent",oldFile+".old")
opv=1
loop
  /* get name of permanent variable */
  pv=array(oldPermanentVariables,opv,1)
  stoploopif pv=""
  /* transfer value from old to new */
  set pv,grabfilevariable(oldFile+".old",pv)
  opv=opv+1
while forever

```

For more information about the **grabfilevariable()** function, see "[Accessing "Dormant" Variables](#)" on page 250.

The procedure above will not work if the old database is not in author mode, since the `dbinfo()` function will not be able to build a list of permanent variables. In that case you must rely on your knowledge of the original database and hard code the permanent variable names, like this:

```
pAreaCode=grabfilevariable(oldFile+".old","pAreaCode")
pDialingPrefix=grabfilevariable(oldFile+".old","pDialingPrefix")
pCallingCard=grabfilevariable(oldFile+".old","pCallingCard")
```

This procedure transfers three permanent variables from the old database to the new: `pAreaCode`, `pDialingPrefix` and `pCallingCard`.

Verifying Database Identity

The procedure listed above for updating the database relies on the user to pick the correct update database. If they pick the wrong database, there will be a big problem. You can use permanent variables to create a database identity system that will permanently identify a database, even if it has been renamed. We recommend creating three permanent variables with the names `dbVendor`, `dbName` and `dbVersion`. Here is an example showing how these variables can be created in the `.Initialize` procedure (see [“.Initialize”](#) on page 382).

```
permanent dbVendor,dbName,dbVersion
dbVendor="ProVUE Development"
dbName="Power Team Phone Book"
dbVersion="2.0"
```

Once these variables have been created they can be used to verify the identity of a database. In the database update routine you can add verification code in between the two `openfile` statements (see [“Updating Database Structure From Another Database”](#) on page 509). This verification code will stop the update if the user selected the wrong database.

```
if dbVendor#grabfilevariable(oldFile+".old",dbVendor) or
    dbName#dbVendor#grabfilevariable(oldFile+".old",dbName)
message "Please pick another database. "+
    "The file you picked is not an update for "+oldFile+"."
closefile /* close the bogus update file */
changenname oldFile /* and restore the original name */
endif
```

You could make this procedure even more robust by adding a check to make sure that the version number of the update file is newer than the version number of the old file.

Database Navigation and Editing

When you are manually working with a database you can use your eyes to see what you are clicking on and modifying. A procedure doesn't have eyes to see with, but it can still navigate and modify the database. Since the procedure can't see what it is doing you have to give it exact instructions to get the job done correctly. Imagine giving directions to a blindfolded person (go 23 paces, turn left, go 14 paces, turn right, etc.) Using a procedure to navigate and edit the database requires the same type of precise instructions.

To illustrate how a procedure can navigate and move around the database we'll use this database of national parks.

Park	Address	City	Sta	Zip	Phone	Fee	URL
Assateague Island National Seashore	7206 National Seashore	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov/gettysburg
Glacier National Park	P.O. Box 128	West Glacier	MT	59936	(406) 888-7800	\$5.00	http://www.nps.gov/glacier
Grand Canyon National Park	P.O. Box 129	Grand Canyon	AZ	86023	(520) 638-2631	\$10.00	http://www.nps.gov/grandcanyon
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.nps.gov/grandteton
Great Basin National Park		Baker	NV	89311	(775) 234-7331	\$0.00	http://www.nps.gov/greatbasin
Great Smoky Mountains National Park	107 Park Headquarters	Gatlinburg	TN	37738	(865) 436-1200	\$0.00	http://www.nps.gov/greatsmoky
Gulf Islands National Seashore	1801 Gulf Breeze Parkway	Gulf Breeze	FL	32561	(850) 934-2600	\$6.00	http://www.nps.gov/gulfislands
Mount Rushmore National Memorial	P.O. Box 268	Keystone	SD	57751	(605) 574-2523	\$0.00	http://www.nps.gov/mountrushmore
Olympic National Park	600 East Park	Port Angeles	WA	98362	(360) 452-4501	\$10.00	http://www.nps.gov/olympic
Rocky Mountain National Park		Estes Park	CO	80517	(970) 586-1206	\$10.00	http://www.nps.gov/rockymountain
White House	1450 Pennsylvania Avenue	Washington	DC	20241	(202) 208-1631	\$0.00	http://www.whitehouse.gov
Yellowstone National Park	P.O. Box 168	Yellowstone	WY	82190	(307) 344-7381	\$10.00	http://www.nps.gov/yellowstone
Yosemite National Park	P.O. Box 577	Yosemite	CA	95389	(209) 372-0200	\$10.00	http://www.nps.gov/yosemite

21 visible/21 total

As shown above we'll start out with the database on the record for [Grand Canyon National Park](#). The current field is the [City](#) field.

Moving Up and Down in the Database

The basic statements for moving the currently active record are [firstrecord](#), [lastrecord](#), [uprecord](#) and [downrecord](#). The [firstrecord](#) statement makes the very first visible record in the database the currently active record (the record at the top of the data sheet).

Park	Address	City	Sta	Zip	Phone	Fee	URL
Assateague Island National Seashore	7206 National Seashore	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley

The **lastrecord** statement makes the very last visible record in the database the currently active record (the record at the bottom of the data sheet).

Rocky Mountain National Park	Estes Park	CO	80517	(970) 586-1206	\$10.00	http://www.r
White House	1450 Pennsylv	Washington	DC	20241	(202) 208-1631	\$0.00 http://www.r
Yellowstone National Park	P.O. Box 168	Yellowstone	WY	82190	(307) 344-7381	\$10.00 http://www.r
Yosemite National Park	P.O. Box 577	Yosemite	CA	95389	(209) 372-0200	\$10.00 http://www.r

21 visible/21 total

The **uprecord** and **downrecord** statements move the currently active record either one record up (towards the top of the data sheet) or one record down (towards the bottom of the data sheet).

To find out if the currently active record is the first or last visible record in the database, use the **info("bof")** and **info("eof")** functions (**bof** stands for **beginning of file** and **eof** stands for **end of file**).

Here's an example that uses the statements and functions described in this section to count the number of parks with no access fee.

```
local freeparks
freeparks=0
firstrecord
loop
  if Fee=0
    freeparks=freeparks+1
  endif
  stoploopif info("eof")
  downrecord
while forever
```


Although this procedure will work, it will also be unnecessarily slow. Avoid scanning through the database whenever possible. Here's another way to write this same procedure that will be much, much faster. For a small database with a couple of dozen records like our example the speed difference isn't too important, but for a large database with thousands of records we're talking about the difference between seconds vs. minutes.)

```
local freeparks
freeparks=0
formulasum freeparks,?(Fee=0,1,0)
```

Another way to reposition the currently active record is to search for something using the `find` statement (see “[Finding Information](#)” on page 552). The `find` statement has one parameter, a formula. Starting from the top of the selected records, Panorama scans down the database until it finds a record that makes this formula true. For example, this procedure will scan down the database until it finds a record where the `Park` field contains `Everglades`.

```
find Park contains "Everglades"
```

Notice that the active field stays the same (`City`) even though the formula searches the `Park` field.



Park	Address	City	Sta	Zip	Phone	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov/gettysburg

After the `find` statement you can check to see if Panorama actually found anything with the `info("found")` function.

To find the next match use the `next` statement (see “[Finding Information](#)” on page 552). This is just like the `find` statement except there is no formula...it re-uses the formula supplied with the `find` statement. You can continue to use the `next` statement over and over again until the `info("found")` function tells you there are no more matches.

Here’s an example that locates all parks with no access fee and deletes them from the database:

```
find Fee=0
loop
  stoploopif (not info("found"))
  deleterecord
  next
while forever
```

Once again, this procedure will work but will be slow. Here’s a faster solution:

```
select Fee<>0
removeunselected
```

Why do I keep showing you these alternate examples? If you are a C or a Pascal programmer you are probably used to solving many problems with loops. In Panorama, a loop is often not the best solution because it is too slow. It may take some research, but you can usually find a Panorama statement that will do the same job much faster.

Moving Left and Right

The basic statements for moving the currently active field are `field`, `left` and `right`. The `field` statement moves directly to the specified field. For example

```
field Park
```

will make the `Park` field the current field.



Park	Address	City	Sta	Zip	Phone	Fee	URL
Assateague Island National Seashore	7206 National Highway 17	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades

If a field name has spaces or other unusual punctuation you must surround it with quotes (see “[Constants](#)” on page 49). Be sure to use quotes and not « » (chevrons).

```
field "Phone Number"
```

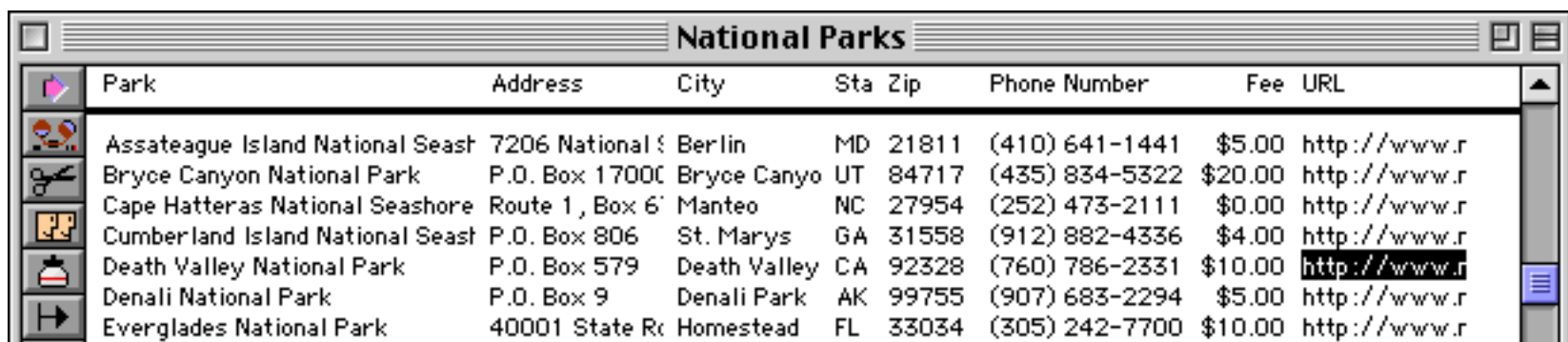


Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 17	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades

You can also use a formula to calculate the field name. If you do so you must surround the formula with (and) parentheses. For example this procedure will move to the last column in the database — any database.

```
field (array(dbinfo("fields",""),arraysize(dbinfo("fields",""),1),1))
```

The formula uses the `dbinfo()` function (see “[Getting Information About Field Structure](#)” on page 505) to calculate the name of the last field (in this case `URL`) and then the field statement jumps to that field.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 17	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades

To move left one field (column in the data sheet) use the **left** statement (see “**LEFT**” on page 5466). To move right one field (column in the data sheet) use the **right** statement (see “**RIGHT**” on page 5679). You can find out if this operation succeeded by using **if stopped**. This will be true if the procedure tried to move to the left of the first column or to the right of the last column. Here is a procedure that scans the entire database and converts every text field to upper case.

```
field (array(dbinfo("fields",""),1,1)) /* move to first field */
loop
  if datatype(info("fieldname"))="Text" /* is this a text field? */
    formulafill upper(«») /* if yes, convert to upper case */
    /* Note: «» is shorthand for current field */
  endif
  right /* move to next field */
until stopped /* stop if we just tried to move past last field */
/* could also use until info("stopped") */
```

Here’s the result of running this procedure on our **National Parks** sample database.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
ASSATEAGUE ISLAND NATIONAL	7206 NATIONA	BERLIN	MD	21811	(410) 641-1441	\$5.00	HTTP://WWW
BRYCE CANYON NATIONAL PARK	P.O. BOX 1700	BRYCE CANY	UT	84717	(435) 834-5322	\$20.00	HTTP://WWW
CAPE HATTERAS NATIONAL SEA	ROUTE 1, BOX 6	MANTEO	NC	27954	(252) 473-2111	\$0.00	HTTP://WWW
CUMBERLAND ISLAND NATIONAL	P.O. BOX 806	ST. MARYS	GA	31558	(912) 882-4336	\$4.00	HTTP://WWW
DEATH VALLEY NATIONAL PARK	P.O. BOX 579	DEATH VALLEY	CA	92328	(760) 786-2331	\$10.00	HTTP://WWW
DENALI NATIONAL PARK	P.O. BOX 9	DENALI PARK	AK	99755	(907) 683-2294	\$5.00	HTTP://WWW
EVERGLADES NATIONAL PARK	40001 STATE F	HOMESTEAD	FL	33034	(305) 242-7700	\$10.00	HTTP://WWW

Notice that this procedure doesn’t reference any specific field names in the **National Parks** database. This procedure will actually work on any database.

Moving “Left” and “Right” on a Form

The examples in the previous section all showed the data sheet, but a procedure can also move to a specific field within a record on a form. To illustrate this we’ll use this form from a **Contacts** database.

Just as when the data sheet is open a procedure can move to a specific field with the **field** statement.

```
field Zip
```

Panorama will jump to the specified field.

The screenshot shows a form titled "Contacts:Person (2 Column)". The form is divided into two columns. The left column contains fields for Name (Sam), Title, Company, Address (6051 Pheasant), and Country (Inverness, IL, 60067). The right column contains fields for Phone, Fax, Email, and Notes. The "Country" field is currently selected, indicated by a blue highlight. The status bar at the bottom shows "103 visible/103 total".

The `left` and `right` statements don't move left and right on the form, but move to the next column based on the order of the fields on the data sheet. For example, suppose you start on the `First` field.

The screenshot shows the same form as above. The "Name" field (Sam) is now selected, indicated by a blue highlight. The status bar at the bottom shows "103 visible/103 total".

The `right` statement will cause Panorama to jump to the next field to the right in the data sheet. In this case that field (`Last`) is also to the right on the form.

The screenshot shows the same form as above. The "Pack" field is now selected, indicated by a blue highlight. The status bar at the bottom shows "103 visible/103 total".

The next field to the right in the data sheet, `Credit Card`, does not exist on this form, so at this point the current field selection is invisible. The `Credit Card` field is the current field, however, and would be modified if the procedure used a statement like `formulafill` at this point.

The screenshot shows the same form as above. The "Name" field (Sam) is now selected, indicated by a blue highlight. The status bar at the bottom shows "103 visible/103 total".

The next field to the right in the data sheet, **Title**, does have a data cell on this form.

The procedure can continue moving to the “right” until it gets to the last column in the data sheet. The **left** statement moves in the opposite direction.

Moving to an Empty Line Item Field

Line items are used for repeating items within a record (see “[Repeating Fields \(Line Items\)](#)” on page 222). When creating a new line item line in a procedure you will want to move to the first empty line item field. To illustrate this, consider this simple invoice form.

To add a new line item to this invoice the procedure must move to the **Quantity7** field. One way to do this would be with a loop.

```

local n
n=1
loop
  field ("Quantity"+str(n))
  stoploopif «»=""
  n=n+1
while forever

```

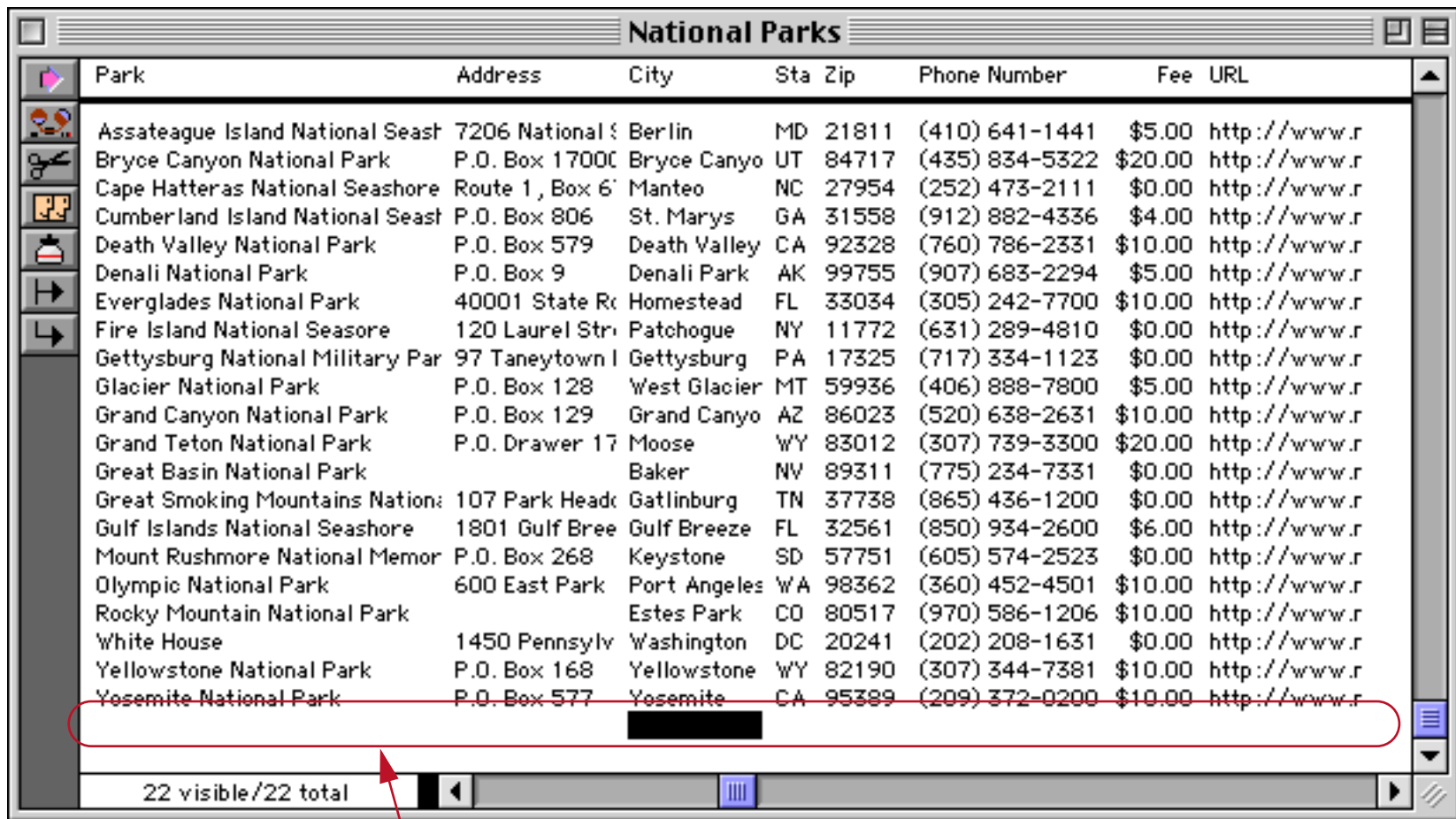
Since this is such a common operation when working with line item fields Panorama has a built in statement to do this job. The statement is called `emptyfield` (see “[EMPTYFIELD](#)” on page 5191). The `emptyfield` statement has one parameter, the name of the line item field to jump to. This field name must be surrounded with quotes and must be followed by the Ω character (see “[Special Characters](#)” on page 57).

```
emptyfield "Quantity $\Omega$ "
```

If there aren't any empty line item fields this statement simply leaves the current field wherever it was.

Adding and Deleting Records

To add a new record at the end of the database use the `addrecord` statement.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov/gettysburg
Glacier National Park	P.O. Box 128	West Glacier	MT	59936	(406) 888-7800	\$5.00	http://www.nps.gov/glacier
Grand Canyon National Park	P.O. Box 129	Grand Canyon	AZ	86023	(520) 638-2631	\$10.00	http://www.nps.gov/grandcanyon
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.nps.gov/grandteton
Great Basin National Park		Baker	NV	89311	(775) 234-7331	\$0.00	http://www.nps.gov/greatbasin
Great Smoky Mountains National Park	107 Park Headquarters	Gatlinburg	TN	37738	(865) 436-1200	\$0.00	http://www.nps.gov/greatsmoky
Gulf Islands National Seashore	1801 Gulf Breeze Parkway	Gulf Breeze	FL	32561	(850) 934-2600	\$6.00	http://www.nps.gov/gulfislands
Mount Rushmore National Memorial	P.O. Box 268	Keystone	SD	57751	(605) 574-2523	\$0.00	http://www.nps.gov/mountrushmore
Olympic National Park	600 East Park	Port Angeles	WA	98362	(360) 452-4501	\$10.00	http://www.nps.gov/olympic
Rocky Mountain National Park		Estes Park	CO	80517	(970) 586-1206	\$10.00	http://www.nps.gov/rockymountain
White House	1450 Pennsylvania Avenue	Washington	DC	20241	(202) 208-1631	\$0.00	http://www.whitehouse.gov
Yellowstone National Park	P.O. Box 168	Yellowstone	WY	82190	(307) 344-7381	\$10.00	http://www.nps.gov/yellowstone
Yosemite National Park	P.O. Box 577	Yosemite	CA	95389	(209) 372-0200	\$10.00	http://www.nps.gov/yosemite

new record added at end of database

To insert a new record just above the current record use the `insertrecord` statement. For example, suppose you start with the database on the [Death Valley National Park](#) record, like this.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland

The `insertrecord` statement inserts a record just above Death Valley.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
[REDACTED]							
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades

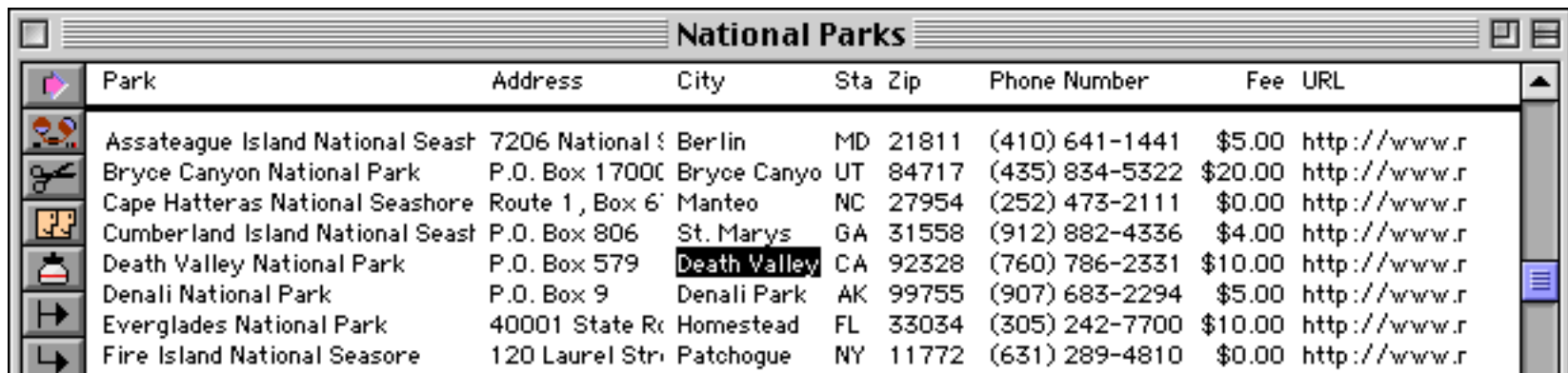
To insert a new record just below the current record use the `insertbelow` statement. For example, suppose you start with the database on the [Death Valley National Park](#) record, just like the previous example.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland

The `insertbelow` statement inserts a record just below Death Valley. Notice that it also moves the current field to the first field in the database.

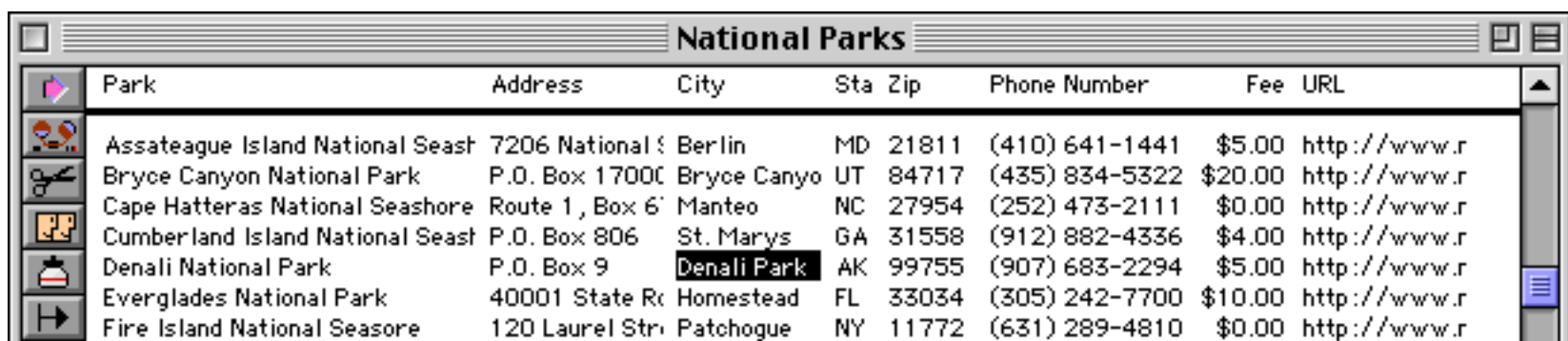
Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
[REDACTED]							
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades

To delete the current record use the `deleterecord` statement. Once again we'll start with the database on the [Death Valley National Park](#) record.



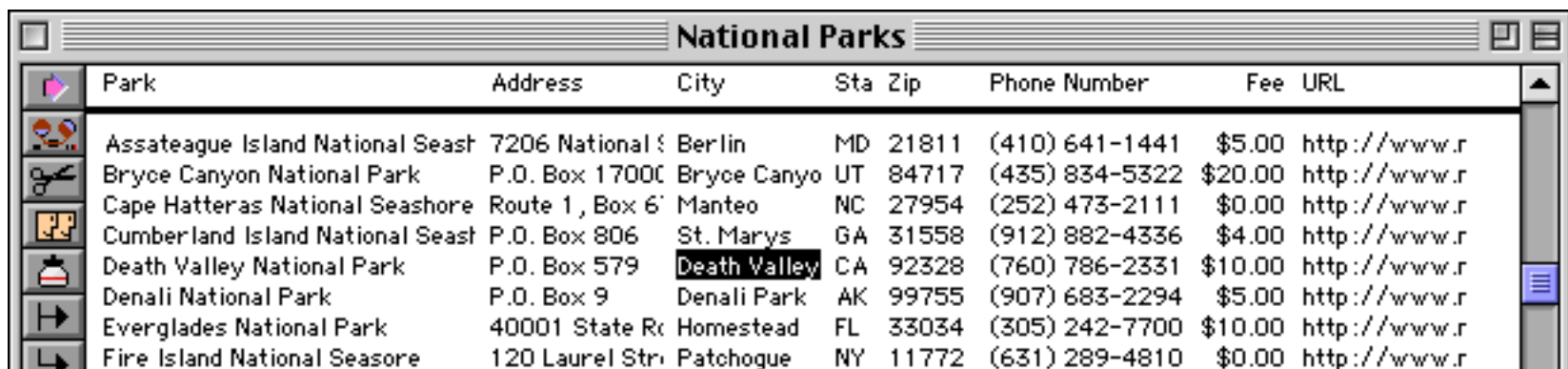
Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland

The `deleterecord` statement deletes [Death Valley](#) and makes the record below [Death Valley](#) ([Denali Park](#)) the current record.



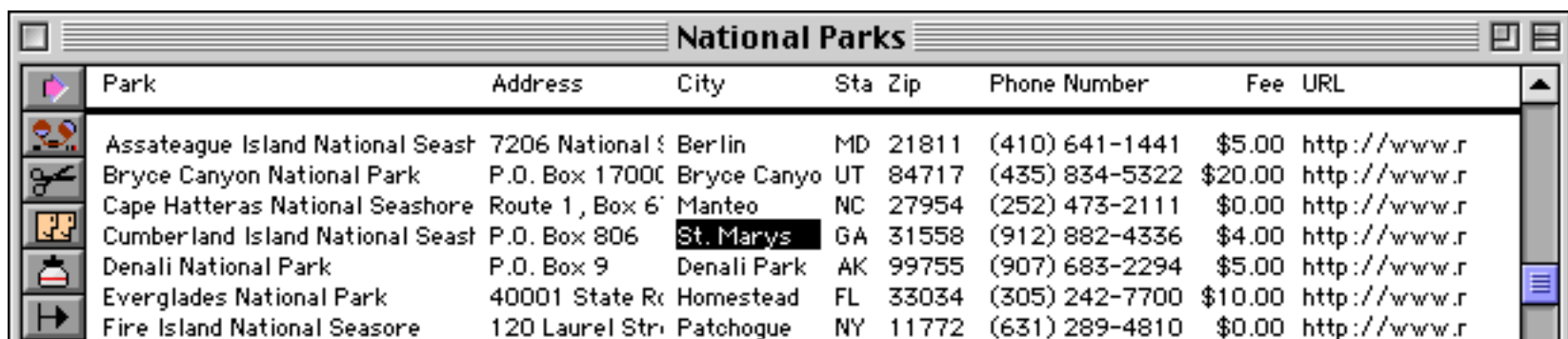
Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland

Another way to delete the current record use the `deleteabove` statement. Once again we'll start with the database on the [Death Valley National Park](#) record.



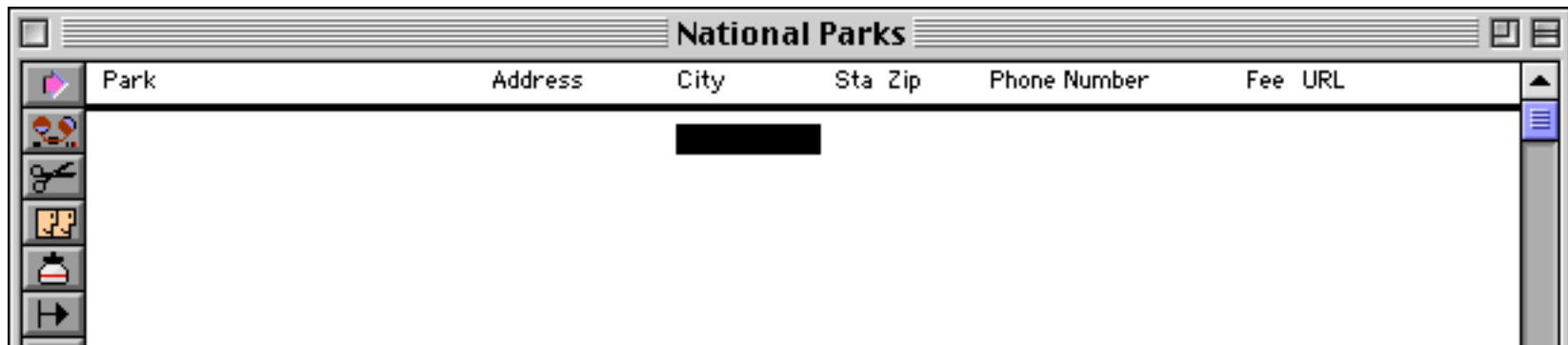
Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland

The `deleteabove` statement deletes [Death Valley](#) and makes the record that was above [Death Valley](#) ([Cumberland Island National Seashore](#)) the current record.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 61	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland

The `deleteall` statement deletes all the data in the entire database, leaving just one blank record.



Needless to say you need to be very careful with this statement!

Modifying the Database One Cell at a Time

To modify an individual data cell in the current record (see “[Moving Up and Down in the Database](#)” on page 512) you use an assignment statement (see “[Assignment Statements](#)” on page 243). Unlike every other statement, an assignment statement has no specific keyword that identifies the statement. Assignment statements always have the format shown below:

```
<data storage location> = <formula>
```

The first part of the assignment statement is the **data storage location**. This is the final destination for the data that is being moved. In fact, sometimes the data storage location is simply called the destination of the assignment. The data storage location may be a variable, a field in the currently active record, or the clipboard.

The next part of the assignment statement is the equals symbol. This identifies this statement as an assignment statement.

After the equals symbol is the **formula**. The formula may simply take a variable or field and pass it along, or it may process, calculate or filter the data before it passes it along to be stored in the data storage location. Here’s a simple assignment statement that takes the contents of **B** and moves it into **A**. After this statement is finished both **A** and **B** will contain the same value.

```
A=B
```

More complicated assignment statements may combine multiple fields or variables, and they may process the data in some way. An assignment statement may also take a constant value and store it. Here are some examples:

```
A=B*C
```

```
Name=upper(myName)
```

```
City="San Francisco"
```

In each case, the process is the same. First Panorama calculates the formula to produce a data value. Then it stores the data value in a data storage location.

Accessing and Modifying the Current Cell

To access or modify the current cell use `<>` (see “[Special Characters](#)” on page 57). For example this statement sets the current cell to **Bingo**.

```
<>="Bingo"
```

This procedure stores the contents of the current cell in the variable **OriginalData**.

```
OriginalData=<>
```

To find out which field is the current cell use the `info("fieldname")` function.

Accessing and Modifying the Clipboard

In addition to variables and fields, the operating system itself provides one spot for stashing data...the **clipboard**. This is where data goes when you use the **Copy** or **Cut** commands in the Edit menu. To grab any text that is in the clipboard, use the `clipboard()` function. To put data on the clipboard, use an assignment with `clipboard` on the left hand side of the equals sign.

The example below takes an address, formats it and copies it onto the clipboard.

```
clipboard=Name+¶+Address+¶+sandwich(" ",City," ")+State+" "+Zip
```

Here's an example that grabs a name from the clipboard and selects all the records containing that name:

```
local findThis
findThis=clipboard()
select Name contains findThis
```

This example copies the contents of the clipboard into a variable before using it in the `select` statement (see “[Selecting Information](#)” on page 557). This is not absolutely necessary—in fact this procedure could have been written in a single line like this:

```
select Name contains clipboard()
```

However, the original procedure will be much faster. Because of the overhead involved in querying the operating system, accessing the clipboard is much slower than accessing a variable. If you're only going to be accessing the clipboard a few times, by all means use it directly. But if you are going to access the clipboard over and over again (as the `select` statement does) it's much better to copy the value into a variable first.

Triggering Automatic Calculations

A database can be set up so that when a field is modified by the user, one or more formulas is automatically calculated (see “[Automatic Calculations](#)” on page 303 of the *Panorama Handbook*). When an assignment statement modifies a field, however, these formulas are not automatically calculated. This is to give the procedure programmer the ultimate control over all calculations that occur during the procedure.

If you as the programmer would like the automatic calculations to be performed during an assignment, add an extra equal symbol to the assignment. The two equal symbols must be adjacent with no spaces between them, like this:

```
PriceΩ==19.95
```

In this example, storing the value 19.95 will most likely trigger several additional calculations to compute the total for this line item and the total for the entire invoice.

Triggering Automatic Procedures

A database can be set up so that when a field is modified by the user, a procedure is automatically triggered. This may be a specific procedure for this field (see “[Automatically Triggering a Procedure](#)” on page 314), or the generic **.ModifyRecord** procedure (see “[.ModifyRecord](#)” on page 383). These procedures are never triggered automatically when an assignment statement modifies a field, even when the double equal assignment is used (see previous section). If you want a procedure to be triggered after a field is modified, you must call the procedure explicitly with the `call` statement (see “[Subroutines](#)” on page 261). This example modifies several fields, then calls the **.ModifyRecord** procedure:

```
City="San Francisco"
State="CA"
<Area Code>="415"
call .ModifyRecord
```

If the automatically triggered procedure expects that a certain field is active when it is triggered you should make sure that field is active by using the field statement before calling the procedure.

The Set Statement

The **set** statement performs the same job as an assignment statement—moving a data item from one place to another. However, unlike the assignment statement, the data storage location is not known in advance. Instead, the data storage location is calculated using a formula.

The set statement has two parameters:

```
set <data storage location formula>,<formula>
```

The first parameter is a formula that calculates the name of the data storage location. Suppose you wanted to store the data in a field named **Widget**. In an assignment you would simply use this name, but in the **set** statement the name must be calculated, in this case **"Widget"**. (Of course this is a silly example, because if we knew the field name in advance we might as well use a regular assignment statement. We'll look at a better example in a minute.)

The second parameter is the formula. This formula produces the data that will be stored in the data storage location. It's exactly the same as the formula used in an assignment statement.

Here's our example. Suppose we have a database where each record has 10 phone number fields, **Phone1**, **Phone2**, **Phone3**, etc. We want to write a procedure that will automatically store a new phone number in the first empty field. Here's one way to get this job done using the **set** statement:

```
local newPhone,Counter,tempPhone
Counter=1 newPhone=""
gettext "New phone #:",newPhone
loop
  tempPhone=grabdata("","Phone"+str(Counter))
  if error
    message "No empty phone number slots!"
    stop
  endif
  stoploopif tempPhone=""
  Counter=Counter+1
while forever
  set "Phone"+str(Counter),newPhone
```

The procedure starts by initializing the variables, and asking the user to input the new phone number. Then it loops through the phone number fields, starting with **Phone1**, then **Phone2**, etc. It checks each field to see if it is empty. If it is, the loop stops and the new phone number is stored with the **set** statement. The first parameter of the **set** statement calculates the field name with the formula **"Phone"+str(Counter)**. For example, if the fourth phone number field was empty, this formula will calculate the field name **Phone4**.

The FormulaCalc Statement

The **formulacalc** statement is similar to the set statement. It's different from the **set** statement because the data storage location is known in advance, but the formula is not known in advance. Instead, the formula itself is calculated using another formula. The **formulacalc** statement has two parameters:

```
formulacalc <data storage location>,<formula>
```

The first parameter is the **data storage location**. This should be a variable or field name, just as in the regular assignment statement.

The second parameter is the **formula**. This formula must itself produce another formula, which is then calculated to produce the data that will be stored in the data storage location. Usually there is a field or variable that contains the formula you want to calculate.

Our example of the `formulacalc` statement lets the user type in a formula, then calculates the formula and displays the result.

```
local xFormula, Answer
xFormula=""
gettext "Enter the formula", xFormula
formulacalc Answer, xFormula
message Answer
```

The `formulacalc` statement was primarily designed to make it easy to build a calculator with Panorama. I'm sure some enterprising programmer out there will find some other uses as well.

Opening the Input Box

The data sheet and data cells in a form use a pop-up input box for editing data (see "[The Input Box](#)" on page 272 of the *Panorama Handbook*). The `editcell` statement automatically opens the input box for the current cell. For example, this procedure adds a new record and automatically opens the input box to get ready for data entry.

```
addrecord
editcell
```

When used on a data sheet window the result of this procedure looks like this.

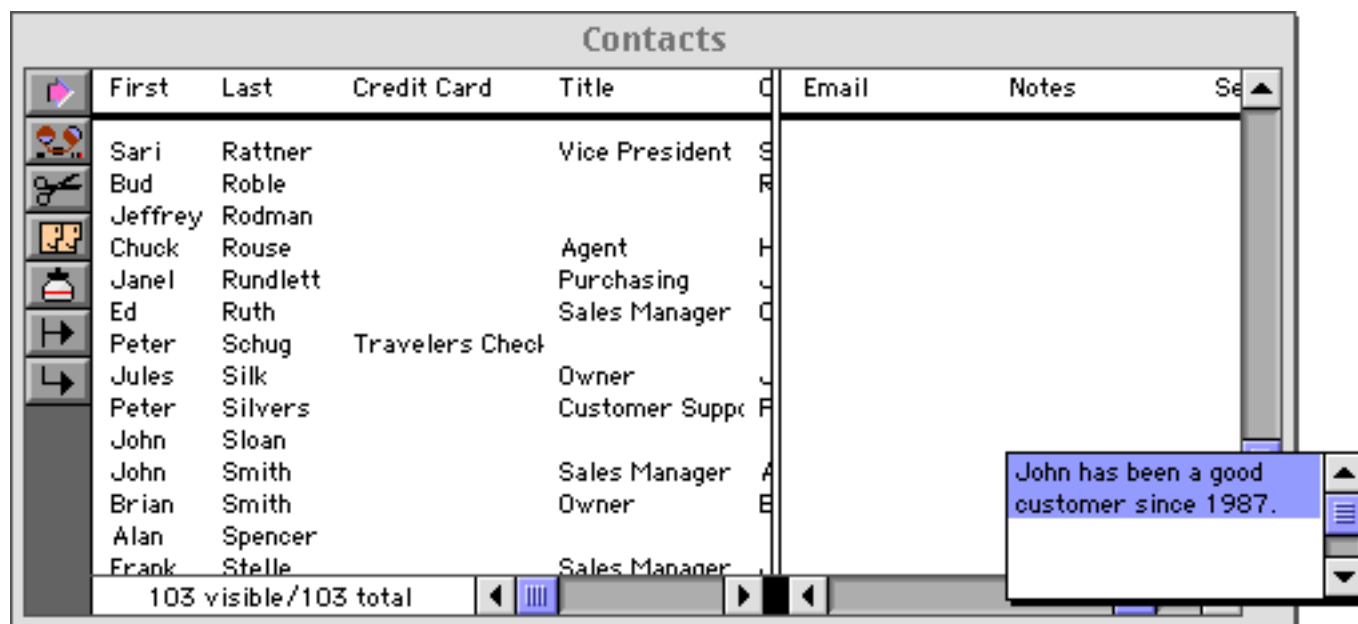
First	Last	Credit Card	Title	Company	Address
Lee	Tucker		Sales Manager	Latham Video	4792 Latham
Pat	Turner		President	P.T. Plumbing	1009 Secret Bay
Steve	West			S.W. Plumbing	1175 Wilson Rd
Karen	Wilson		Vice President	Evanston Lumber	498 Noyes
Jerry	Wilson				3050 North Main
Gregory	Wing		President	GW Printing	779 Arnold Rd
William	Wong	Cash			7292 Delvin Wy
Raymon	Wood				5420 W Crosby
Victor	Yalom		Purchasing	San Francisco Lumber	854 14th St
Peter	Yarensky		Owner	Peter's Appliances	41 Elm St

input box open and ready for data entry in new record

We recommend that the `editcell` statement be the last statement in a procedure. If it is not the last statement in the procedure you should use the `editcellstop` statement, like this.

```
if info("trigger") contains "Add Record"
  addrecord
  editcellstop
endif
```

If the cell contains data the `editcell` statement normally selects all of the data when it opens the input box, like this.



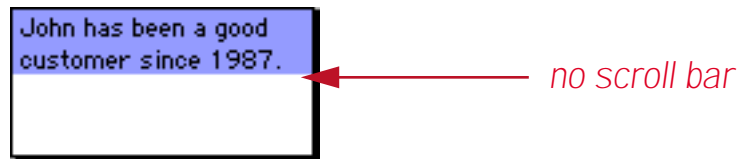
The `editselect` statement allows a procedure to control what text is selected. It must be used immediately before the `editcell` or `editcellstop` statement. The `editselect` statement has two parameters: `start` and `end`. Both parameters are numbers from 0 (first character) to 32768 (last character). The table below shows the effect of different parameters with this statement.

Code	Result
<code>editselect 0,0</code> <code>editcell</code>	
<code>editselect 32768,32768</code> <code>editcell</code>	
<code>editselect 0,32768</code> <code>editcell</code>	
<code>editselect 5,8</code> <code>editcell</code>	

The input box normally has a scroll bar when it is more than about an inch high. The `noeditscroll` statement suppresses the scroll bar no matter how high the input box is. This statement is designed to be used as a prefix for the `editcell` and `editcellstop` statements, like this.

```
noeditscroll
editcell
```

The input box appears without a scroll bar.

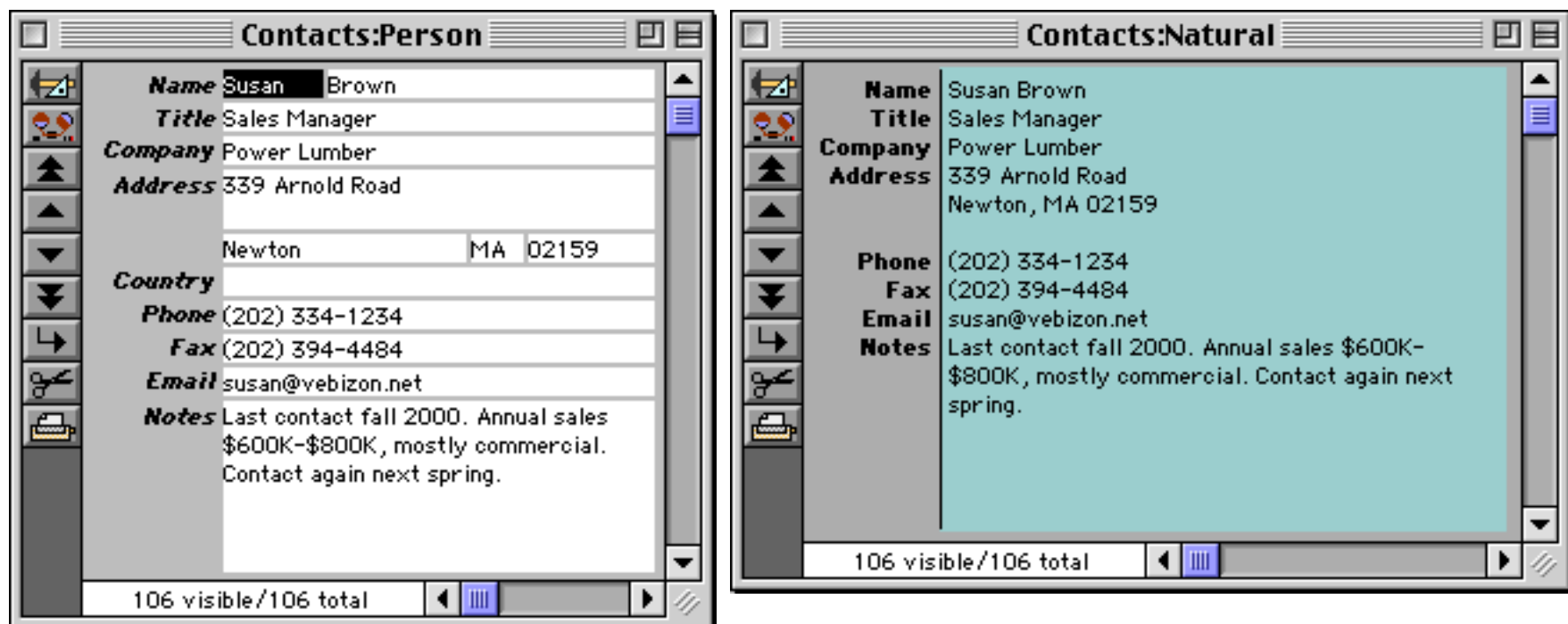


In a form window a procedure can even open an input box in “thin air” in an arbitrary location. No data cell object on the form is required. To learn how to do this see “[FLOATINGEDIT](#)” on page 5257 of the *Panorama Reference*.

These statements work only with data cells. To learn how to control editing within a Text Editor SuperObject see “[Text Editor SuperObject Commands](#)” on page 670.

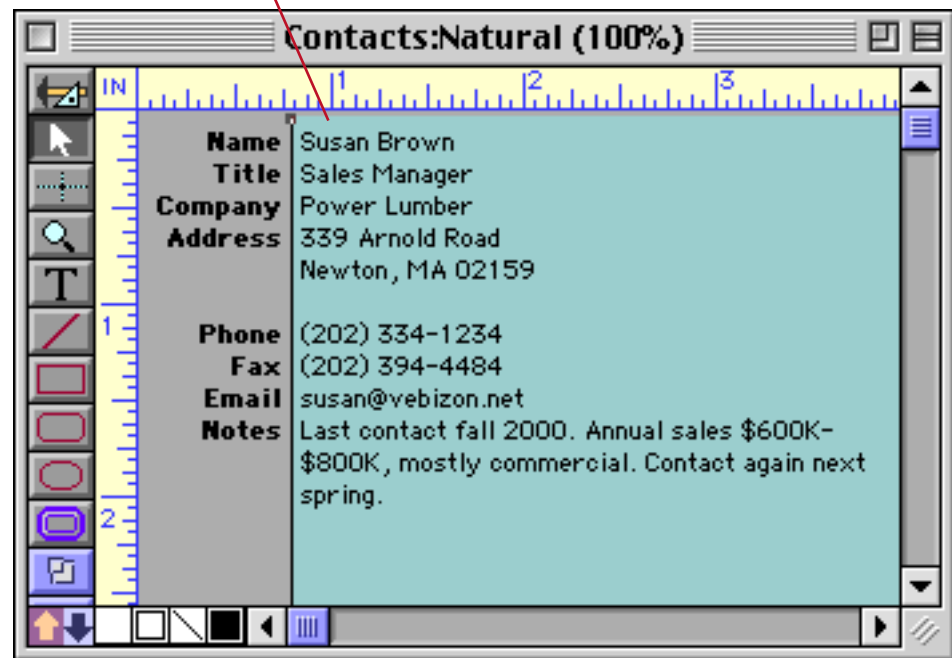
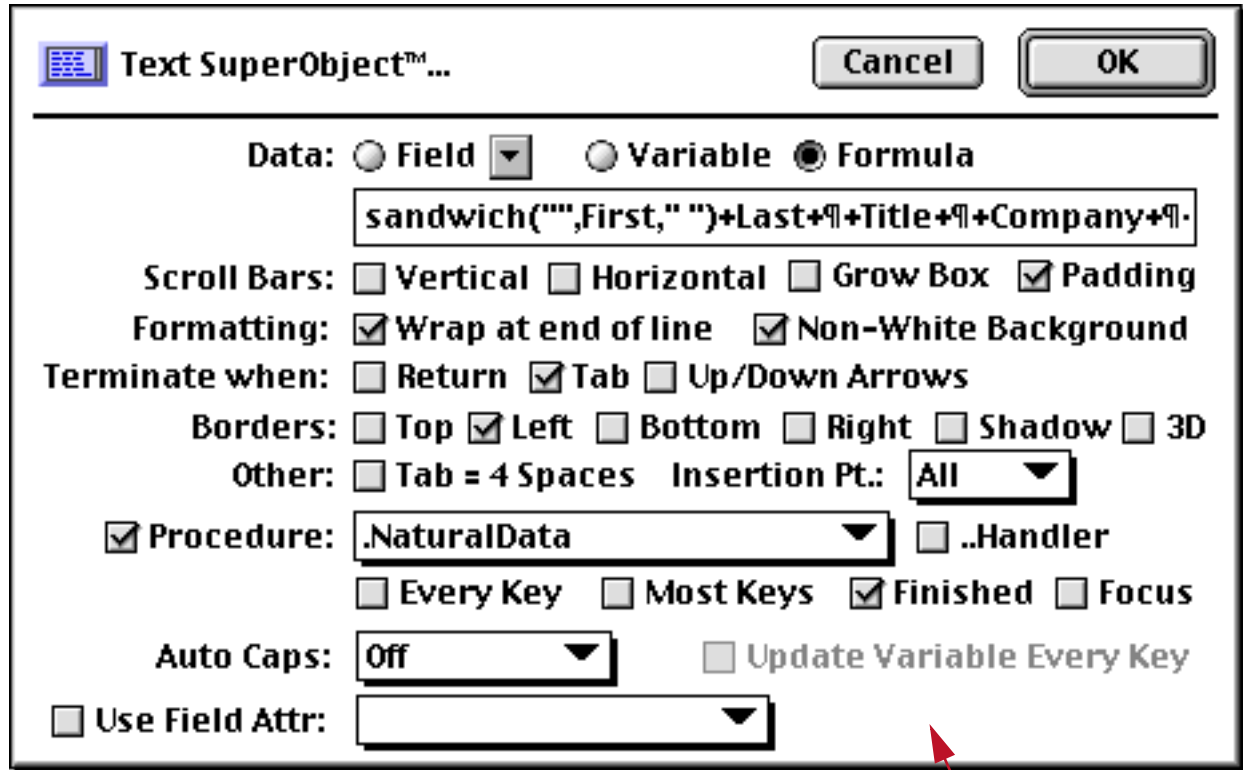
“Natural” Data Entry

Computers and people often don’t think alike. Computers tend to use rigid formats, while people like to be more free-form. In the case of databases with contact information it’s best to store lots of separate fields for first and last names, street address, city, state, zip etc. (as shown on the left below). This gives the most flexibility in sorting, selecting and reporting data. However, from a data entry point of view it would be much nicer to enter data in a more natural format (as shown on the right).



Natural Data Display

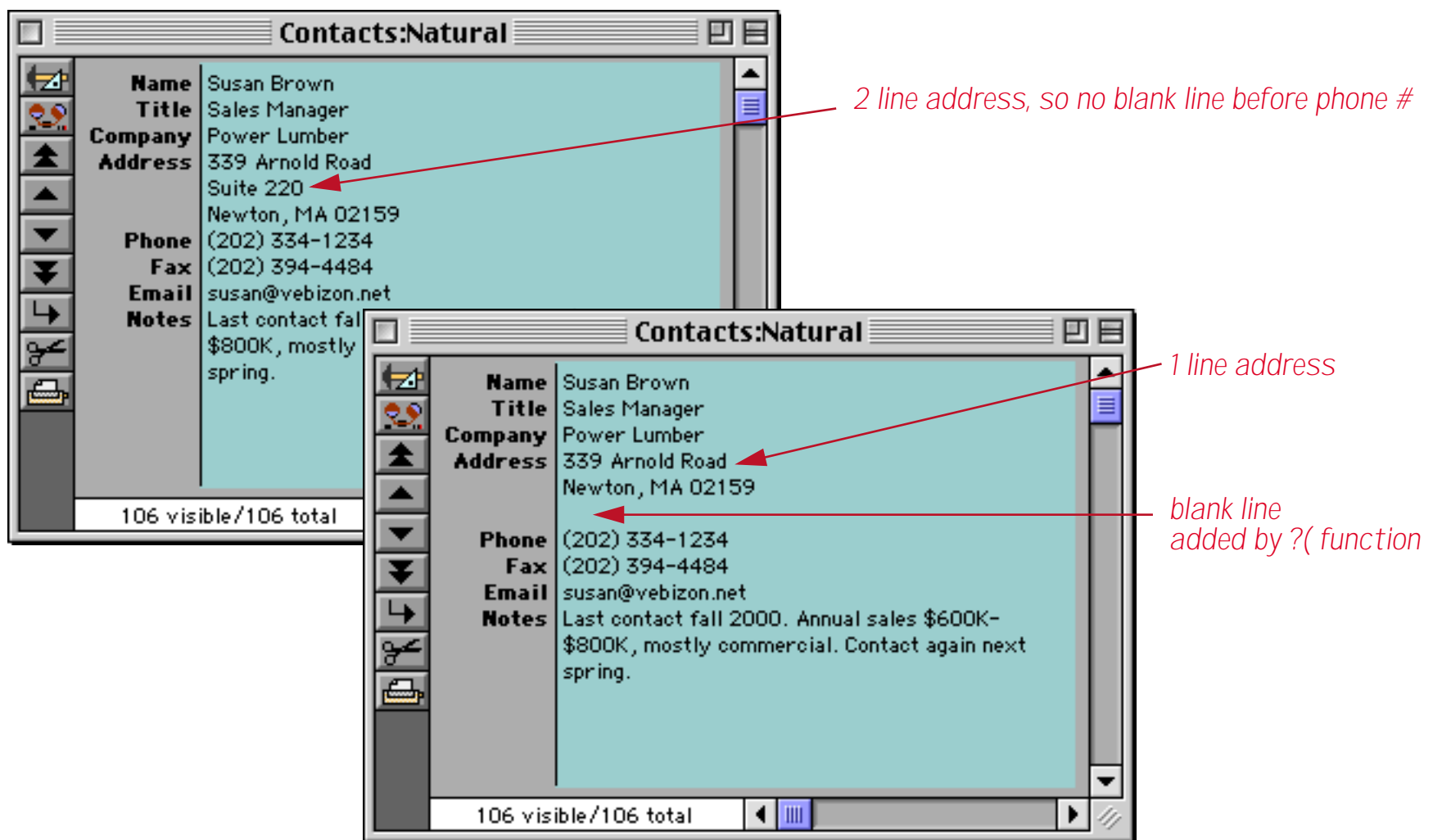
To display data in a natural format use a Text Display SuperObject (see “[Text Display SuperObjects™](#)” on page 608 of the *Panorama Handbook*) or a Text Editor SuperObject (see “[Text Editor SuperObject](#)” on page 639 of the *Panorama Handbook*) with the **Formula** option enabled (see “[Text Editor Options](#)” on page 643 of the *Panorama Handbook*). Of course if you want to be able to edit the data you’ll have to use the Text Editor object. Here is the configuration dialog for the Natural format text editor shown in the preceding section. (Note: The blue-green background behind the text was created with a rectangle object placed behind the Text Editor.)



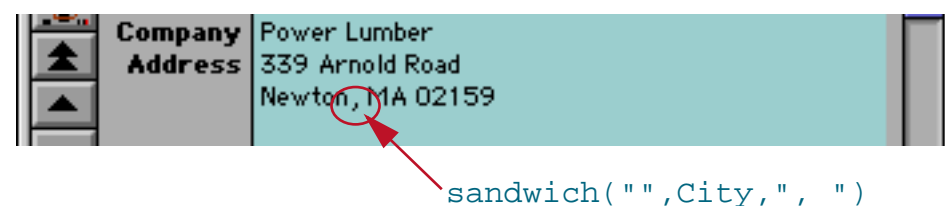
Here is the complete formula for this object.

```
sandwich(" ",First," ") + Last + ¶ +
Title + ¶ +
Company + ¶ +
arrayrange(Address,1,2,¶) + ¶ +
sandwich(" ",City," ") + sandwich(" ",State," ") + Zip + sandwich(" ",Country," ") + ¶ +
?(Address notcontains ¶,¶,"") +
Phone + ¶ + Fax + ¶ + Email + ¶ + Notes
```

The **arrayrange()** function is used to extract a maximum of two lines of address (so if an address has 3 or more lines the extras will be removed). The **?(** function (see “[The ? Function](#)” on page 130) checks to see if the **Address** field contains only one line, and if so, adds an extra blank line between the address and the phone number.



The **sandwich()** functions are used to add punctuation (spaces and commas) only if needed.



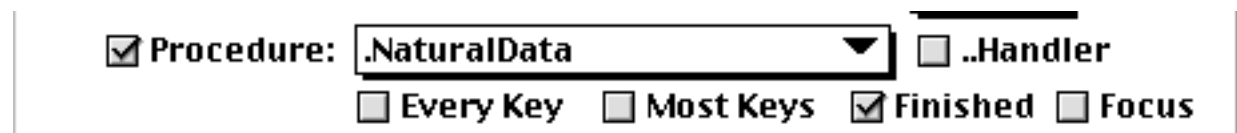
If the city name is empty (for example when entering a new contact) then there is no comma.



Since the box for entering formulas in the Text Editor SuperObject is so small you may want to test out your formula in a Text Display SuperObject first, then copy to a Text Editor once it is working correctly.

Natural Data Entry

Displaying the data in a “natural” format is only half the job. To allow the data to be entered and/or modified in this natural format you’ll need to create a procedure. In our case we’ve configured the Text Editor to trigger a procedure named **.NaturalData** when the **Enter** key is pressed.



Here is the **.NaturalData** procedure itself.

```
local FullContact,FullName,FullAddress,namePrefix,nameMiddle,nameSuffix
local xTitle,xCompany,xPhone,xFax,xEmail,zPhone,zFax,xNotes
FullContact=TextEditingResult
splitlines FullContact,
  "1W",FullName,
  "1W",xTitle,
  "1W",xCompany,
  "3W",FullAddress,
  "1",xPhone,
  "1",xFax,
  "1",xEmail,
  "0",xNotes

Title=xTitle Company=xCompany Email=xEmail Notes=xNotes
getname FullName,namePrefix,First,nameMiddle>Last,nameSuffix
getaddress FullAddress,Address,City,State,Zip,Country
getphone xPhone,"714",Country,zPhone,1,""
Phone=strip(array(zPhone,2,":"))
getphone xFax,"714","",zFax,1,""
Fax=strip(array(zFax,2,":"))
```

The procedure starts by creating the temporary variables it needs. Then it uses the **splitlines** statement (see “[SPLITLINES](#)” on page 5786 of the *Panorama Handbook*) to split the incoming data into eight separate components. Most of these components are one line high, but the address contains three lines and the notes contains all of the text after the tenth line. In the process of splitting the text the **splitlines** statement also automatically capitalizes the first letter of each word in the name, title, company and address. However, the capitalization only happens if all of the text is entered in lower case, so **frank rich** will be converted to **Frank Rich**, but **Scott McBride** must be typed in like that, not as **scott mcbride**.

The **getname** statement splits the name into five separate components (see “[GETNAME](#)” on page 5306 of the *Panorama Reference*). This database only uses two of the components (first and last names) so the other three components are simply discarded. This table shows a few examples of how a name is split up into its individual components.

Sample	Prefix	First	Middle	Last	Suffix
Frank Rich		Frank		Rich	
Ms. Susan Kay Olson	Ms.	Susan	Kay	Olson	
John Kuttel DVM		John		Kuttel	DVM
General Dwight A. Eisenhower	General	Dwight	A.	Eisenhower	
Mark Jackson Jr.		Mark		Jackson	Jr.

The `getaddress` statement splits the address into five separate components (see “[GETADDRESS](#)” on page 5288 of the *Panorama Reference*). It is primarily designed to handle US and Canadian addresses. If you purchased the optional zip code dictionary (see “[Zip Code Lookup](#)” on page 145) you can enter just the zip code and let Panorama fill in the city and state for you (see the 3rd and 4th examples in the table below). This table shows a few examples of how a name is split up into its individual components.

Sample	Street Address	City	State	Zip	Country
575 Memorial Drive Cambridge, MA 02139	575 Memorial Drive	Cambridge	MA	02139	
445 Hoes Lane Piscataway, NJ 08855-1331	445 Hoes Lane	Piscataway	NJ	08855 -1331	
15180 Transistor Lane 92648	15180 Transistor Lane	Huntington Beach	CA	92648	
400 Seaport Court Suite 100 94063	400 Seaport Court Suite 100	Redwood City	CA	94063	
6733 Mississauga Road Mississauga, ON L5N6J5 canada	6733 Mississauga Road	Mississauga	ON	L5N 6J5	CANADA

The `getphone` statement formats the phone number (see “[GETPHONE](#)” on page 5308 of the *Panorama Reference*) but only if the country name is blank, **USA** or **CANADA**. Our example has been set up to default to the 714 area code, but you can use any default area code you wish.

Sample	Output
5557390	(714) 555-7390
3034491234	(303) 449-1234
(412) 987-3859	(412) 987-3859
7307832x23	(714) 730-7832x23

To see how all of this works together let's add a new record and type in the data. We think you'll agree that typing in the data in this natural format is easier than tabbing from field to field to field.

Contacts:Natural

Name	john warnock
Title	chief executive officer
Company	adobe systems incorporated
Address	345 Park Avenue 95110-2704
Phone	4085366000
Fax	4085376000
Email	ir@adobe.com
Notes	Since John Warnock co-founded Adobe Systems in 1982 with Dr. Charles Geschke, the two have worked closely together to develop a stream of pioneering software products that leverage Adobe's core strengths in graphics, publishing, and electronic document technology.

107 visible/107 total

When the entry is complete press the **Enter** key. The **.NaturalData** procedure will process the entry into the separate fields in the database. Here you see both the natural display format and a form displaying each field separately.

Contacts:Natural

Name	John Warnock
Title	Chief Executive Officer
Company	Adobe Systems Incorporated
Address	345 Park Avenue San Jose, CA 95110-2704
Phone	(408) 536-6000
Fax	(408) 537-6000
Email	ir@adobe.com
Notes	Since John Warnock co-founded Adobe Systems in 1982 with Dr. Charles Geschke, the two have worked closely together to develop a stream of pioneering software products that leverage Adobe's core strengths in graphics, publishing, and electronic document technology.

107 visible/107 total

Contacts:Person (2 Column)

Name	John Warnock	Phone	(408) 536-6000
Title	Chief Executive Officer	Fax	(408) 537-6000
Company	Adobe Systems Incorporated	Email	ir@adobe.com
Address	345 Park Avenue	Notes	Since John Warnock co-founded Adobe Systems in 1982 with Dr. Charles Geschke, the two have worked closely together to develop a stream of pioneering software products that leverage Adobe's core strengths in graphics, publishing, and electronic document technology.
Country	San Jose CA 95110-		

107 visible/107 total

Since the data is stored in separate fields you can easily sort, group or select it any way you want. Nevertheless you still have the convenience of entering and editing it in natural format if you wish. To edit the natural format data simply click on it and edit, then press **Enter** to update the database.

Contacts:Natural

Name	John Warnock
Title	Chief Executive Officer
Company	Adobe Systems Incorporated
Address	345 Park Avenue San Jose, CA 95110-2704

The natural data formats demonstrated in this contacts database can be used with any database you create.





Validating a Credit Card Number

Credit cards have an internal checksum that allows a number to be validated for simple data entry errors (for example missing or transposed digits). The `cardvalidate` statement checks to make sure that a number is a valid credit card number. This statement has two parameters. The first is the card number you want to validate. The second parameter should be a variable. The statement will set this variable to `Ok` or `Error` depending on the card number you submit. This example checks the card number in the field `CCNumber` to see if it is a valid credit card number.

```
local cctemp,ccvalid
cctemp=striptonum(CCNumber)
cardvalidate cctemp,ccvalid
if ccvalid<>"Ok"
    message "This credit card number is not valid!"
endif
```

The `cardvalidate` statement cannot tell whether this card number has actually been issued, what the credit limit is, or any other financial information about the card. It simply provides a basic check for missing or transposed digits within the number. Basically, if this statement says that the number is in error you know for sure that the number is wrong, but if this statement says the number is valid you would still need to check with the issuer to determine if this is a valid card.

By the way, it's easy to determine the type of card from the first digit of the number.

Card	First Digit
	3
	4
	5
	6

Here's another handy tip for testing. You can make a "valid" credit card number by taking the digits 1, 2, 3, and 4 and repeating them four times in any order. For example `1111222233334444`, `3333111122224444` and `4444111133332222` are all valid credit card numbers (of course there aren't really any cards with these numbers, but the checksum is ok).

Moving Data Between Files

Many database applications require multiple database files working together. For example, organizing a company's order entry operations usually requires an invoice file, an inventory/price list file, and possibly a customer file. As orders are entered, data will move back and forth between these files to adjust inventory levels, maintain customer credit records, and so forth.

Sometime you may want to display or use information from another file without actually moving it. For example, you may want to display a customer's current credit balance without actually copying it into the invoice. Panorama gives you the option of actually moving data from place to place or simply displaying or printing it.

Panorama's primary method for moving data between databases is the same method used to move data within a record: the assignment statement. Using the lookup(function (along with several variations on this function) you can turn any assignment statement into a cross database transfer. However, assignments are not the only way to transfer data across files...there are also some special statements that are especially useful for multiple-record transfers.

Warning: Panorama can only move data between database files that are currently open. If a database is not open, you must open it before data transfer can take place.

Cross Database Assignment Statements

The primary method for moving data between databases is assignment statements. If you are reading this chapter straight through this may puzzle you, since earlier we stated that assignment statements always worked on the current record on the current database. For example, consider this statement.

```
A=B+C
```

All three of the elements of this assignment (A, B and C) must be either fields in the current record or variables.

To break the limitation to the current record, the assignment statement must use a special function, called a *transfer function*. Panorama has about a half a dozen transfer functions, but at the basic level they all work the same way...you tell them how to find some data and they go get it.

```
A=transferfunction(<how to find the data>)
```

Each transfer function has a series of parameters that tell it how to find the data you want. The transfer function uses these parameters to find the data and bring it back to the formula.

Except for the fact that transfer functions retrieve data from other database files, these functions are just like any other function. You can use them in combination with other operators and functions. For example, suppose you had a transfer function with parameters designed to retrieve a person's name. You could use this function to construct a greeting for a letter like this:

```
Greeting="Dear "+upperword(transferfunction(<how to find the name>))+", "
```

You can only transfer one item of data per assignment statement. If you want to transfer more than one item of information, you'll have to transfer one at a time (with some special exceptions explained later). The transfer functions are designed so that transferring several items of data from a single record is a very fast operation.

Identifying Data to Move

Moving data is easy. The hard part is explaining to Panorama exactly what the data is you want it to move. Unfortunately, we usually can't simply point at the data and say "there it is, go get it." Most of the next few sections will deal with how to explain where the data is in an unambiguous way so Panorama will understand and obey.

Panorama has two basic ways for identifying data to move: by position and by value. Data that is identified by position really is identified by pointing at it and saying “go get it.” For example, you might ask the user to find an item in the price list and click on it. Once the user has located the item, Panorama can easily move the data somewhere else with the `grabdata()` function.

The second way to identify data to move is by value. When this technique is used, we tell Panorama what we are looking for and it goes and finds it for us. For example, we may want to find the current credit balance for Gargantuan Widget Company. In this case, the value is Gargantuan Widget Company. Panorama will scan the database looking for this value, and if it finds a record containing the value, it moves the credit balance from that record to the currently open database.

Identifying data by value is somewhat complicated because the result is uncertain. There are basically three possible outcomes: 1) The value doesn't exist anywhere in the database, 2) There is one record with this value in the database, or 3) There are many records with this value in the database. To write a successful program for moving data between databases you must decide how your procedure will handle each of these situations.

Transfer Function Parameters

All transfer functions have similar parameters for locating the data to transfer. (Note: The parameters do not necessarily appear in the order presented below.)

The first two parameters listed below, **Target Database** and **Data Field**, are used by all transfer functions. The remaining parameters, **Key Field**, **Key Value**, **Default Value**, **Summary Level** and **Separator** are used only by transfer functions that identify data by value.

Target Database. This parameter is the name of the database that contains the data you want to retrieve. Usually this is a fixed name, so you would simply enter the name surrounded by quotes, for example "Price List" or "Customer File". If the name of the target file may change you could put the name in a variable. By the way, the target database must be open, or the transfer function will not work.

The target database may even be the currently open database. This allows you to transfer data from other records in the database to the current record. To specify the current database use the `info("databasename")` function. (You could simply type in the name of the current file surrounded by quotes, but if the file is ever renamed the transfer function will stop working.)

Data Field. This parameter is the field that contains the actual data you want to retrieve. It must be a field within the target database. For example, if you wanted to look up a person's phone number, this would be a field named something like Phone.

The data field should contain data you don't already know. For example, if you know a company's name and want to know their address, the data field would be **Address**. (The **CompanyName** field would be the key field, see below.)

Each transfer function can transfer one and only one data field. You can't use a formula here like

```
City+" "+State+" "+Zip
```

you must specify the name of a single field in the target database. If you want to transfer multiple fields you'll generally have to use multiple assignment statements and transfer one field at a time.

The type of data the transfer function returns depends on the type of data contained in the data field. If the data field is a text field, the transfer function will return a text value. If the data field is a numeric field, the transfer function will return a numeric value. (Exception: If the transfer function is allowed to return multiple matches, the data is always converted to text.)

Note: Prior to version 3.0, Panorama required that the data field name be surrounded by quotes when used in a transfer function. This is no longer necessary, but is still allowed for compatibility with database files created with older versions. Now you can simply type the field name in without quotes. (However, if the field name contains punctuations or spaces it must, of course, be surrounded by chevrons « and ».)

Key Field. This parameter is the field in the target database that contains the data you already know about. For example, suppose you know someone's name but not their phone number. The **Name** field would be the key field. The transfer function will scan the key field looking for an exact match with the data supplied as the **Key Value** parameter.

Only a single field may be used as the key field, and it must match up with the data in the key value. Suppose you want to look up the phone number for **Joe Smith**, and your target database contains separate fields for first and last name. You've got a problem, because your key value does not match up with any one field in the target database. The possible solutions include: 1) matching up on the last name only, 2) creating a new, redundant field in the target database that contains both first and last names, or 3) using the **arraybuild** statement instead of an assignment statement to transfer the data. (The **arraybuild** statement allows any formula to be used as a match, so you are not limited to a single key field and a single key value. Using the **arraybuild** statement to transfer data between files will be described in detail in a later section.)

Note: Prior to version 3.0, Panorama required that the key field name be surrounded by quotes when used in a transfer function. This is no longer necessary, but is still allowed for compatibility with database files created with older versions. Now you can simply type the field name in without quotes. (However, if the field name contains punctuations or spaces it must, of course, be surrounded by chevrons « and ».)

Key Value. This parameter is the value that identifies the data you want to retrieve. For example, if you are looking up data in a price list database, the key value would be values like **180 ohm resistor** or **3/8" lock washer**. The key value must exactly match a value in the key field. If the key field contains a part called a **250 µf capacitor** the key value must be exactly that, not **250 µf Capacitor** or **250 µf capacitor (50V)**.

The key value is usually assembled from one or more fields in the current database. Unlike the key field, the key value may be assembled with any formula you want to use. For example, suppose your target database contains names in a single field (for example **Maureen Livingston** or **Steve Toyota**) but the current database has separate fields for first and last names. No problem here, just use a formula like **First+" "+Last** for the key value. Remember, the key value is a value, so any formula may be used to calculate it. The key field is a field, and must be a single field in the target database.

Default Value. What if the transfer function fails to locate the key value anywhere in the target database? There is no data to retrieve, but the transfer function must return some value. This is the job of the default value. The default value steps in when the requested data is not available for any reason.

The default value should have the same data type as the data field. If the data field is text, the default value should be text. If the data field is numeric, the default value should be numeric. The most common default values are "" (empty string) for text, and 0 for numeric. However, you can use any value you want.

Sometimes you may want to create an assignment that moves data from another database to a field in the current database, but leave that field untouched if the data cannot be found. In this case, the default value should be the field itself. That way if the data is not found Panorama will simply move the current field to itself, essentially leaving it untouched. The partial example below shows how this works. This transfer function looks up a phone number, but if it is not found, uses the current phone number as the default.

```
Phone=transferfunction(...,Phone,...)
```

Summary Level. Panorama normally scans all records in the target database looking for a match between the key value and the key field. However, if you set the summary level parameter to a non-zero value, the transfer function will only scan summary records. The summary level parameter should be a number from 0 to 7. If the value is set to 2, only summary levels 2 and above will be scanned, etc. Unless you specifically want to work with summary records, set this parameter to zero.

Separator. This parameter is only used by transfer functions that can return more than one match at a time, like **lookupall()**. Each returned data item is separated by the text in this parameter. Common separators include commas, slashes, carriage returns (¶) and tabs (→). For example, suppose you want a transfer function to return a comma separated list of all the invoice numbers for a certain customer. The separator is a comma character (",").

Single Record Transfer Functions

Enough theory, let's get down to specifics. The transfer functions described below locate a single record in a database and retrieve one item of data from that record.

grabdata(target database,data field)

The **grabdata()** function identifies data by position. This function retrieves data from the current record in the specified database. For example, if the user has selected **7404 hex inverter** in the Price List database by clicking on it (or searching for it with the **Find/Select** command), this procedure will copy the information into the current line item in the Invoice database.

```
ItemΩ=grabdata("Price List",Part)
PriceΩ=grabdata("Price List",Price)
```

The drawback of the **grabdata()** function is that the user must manually locate the data, but sometimes this is exactly what you want.

A sometimes useful trick with the **grabdata()** function is to grab data in the current database. This allows you to specify the field name with a variable.

lookup(target database,key field,key value,data field,default value,summary level)

The **lookup()** function identifies data by value. When this function is used, Panorama scans every record in the target database starting from the top (including unselected records). When it finds the first match between the key value and the key field, it stops scanning and returns the value in the data field of that record. There may be 1, 10, 100 or 1,000 possible matches, but the **lookup()** function will only return the first one.

For our example, suppose we have two databases: Invoices and Customers. These two databases have 5 identical fields: Company, Address, City, State and Zip. When the user enters the company name, we want to write a procedure that will automatically move the address, city, state and zip from the customer file into the new invoice.

```
gettext "What company name?",Company
Address=lookup("Customers",Company,Company,Address,"",0)
City=lookup("Customers",Company,Company,City,"",0)
State=lookup("Customers",Company,Company,State,"",0)
Zip=lookup("Customers",Company,Company,Zip,"",0)
```

Since each **lookup()** statement can only transfer one value, four lookups are required. Panorama doesn't actually scan the Customer file four times. When a procedure performs multiple lookups with the same target database, key field, and key value Panorama realizes that it doesn't have to re-scan the database—it already knows where the data is, and it just goes and gets it. (If your database is set up for it, you can use the **speedcopy** statement to move this data even faster. See the description of this statement later in this chapter.)

lookuplast(target database,key field,key value,data field,default value,summary level)

The **lookuplast()** function identifies data by value. When this function is used Panorama scans every record in the target database, but unlike the **lookup()** function, this function starts from the bottom and scans up (including unselected records). When it finds a match between the key value and the key field, it stops scanning and returns the value in the data field of that record. There may be 1, 10, 100 or 1,000 possible matches, but the **lookuplast()** function will only return the last such match in the file. (However, there is an exception. If you are scanning the current database, the **lookuplast()** function will skip the current record. This allows you to find the last match in the file not including the current record.)

When the key field and key value may match many times, we are often more interested in the last match instead of the first match. For example, if a customer has many invoices with us, we are probably more interested in the most recent invoice than in the first invoice from long ago. (Of course this is assuming the database is sorted in chronological order!) The `lookuplast()` function allows us to locate this most recent information.

Our `lookuplast()` example finds the most recent order for a given customer, and the amount of that order.

```
local theCustomer,lastOrderDate,lastOrderAmount
theCustomer=""
gettext "Customer name:",theCustomer
lastOrderDate=lookuplast("Invoice",Company,theCustomer,Date,0,0)
if lastOrderDate=0
    message "No previous invoices for this customer."
    stop
endif
lastOrderAmount=lookuplast("Invoice",Company,theCustomer>Total,0,0)
message theCustomer+"'s most recent order was for "+
    pattern(lastOrderAmount,"$#,##")+ " on "+
    datepattern(lastOrderDate,"Month dnth, yyyy")+ "."
```

Here's another example that combines `lookup()` and `lookuplast()`. This example first tries to look up a customer in the Customers database. If they are not found there, it checks to see if there is a previous invoice for this customer.

```
Address=lookup("Customers",Company,Company,Address,"",0)
if Address=""
    City=lookup("Customers",Company,Company,City,"",0)
    State=lookup("Customers",Company,Company,State,"",0)
    Zip=lookup("Customers",Company,Company,Zip,"",0)
else
    Address=lookuplast(info("database"),Company,Company,Address,"",0)
    City=lookuplast(info("database"),Company,Company,City,"",0)
    State=lookuplast(info("database"),Company,Company,State,"",0)
    Zip=lookuplast(info("database"),Company,Company,Zip,"",0)
endif
```

Remember, the `lookuplast()` function locates the matching information that is physically closest to the bottom of the database. What this proximity to the bottom means depends on how the database is sorted.

lookupselected(target database,key field,key value,data field,default value,summary level)

The `lookupselected()` function identifies data by value. When this function is used, Panorama scans every selected record in the target database starting from the top. Unlike a regular `lookup()`, this function skips unselected records.

There are two possible advantages to using the `lookupselected()` function. If the target database contains only a small percentage of selected records compared to unselected records, `lookupselected()` will be faster than `lookup()`. (Of course if the data you want to locate is not selected, `lookupselected()` won't find a match.)

The primary advantage to using `lookupselected()` is the ability to control exactly what records are scanned as part of the lookup. This is especially useful if there is a possibility of more than one match between the key value and key field.

table(target database,key field,key value,data field,default value,summary level)

The `table()` function is the only transfer function that does not require an exact match between the key value and the key field. If it does not find an exact match, the `table()` function will accept a match that is merely "close enough." The table function is designed to be used with rate lookup tables like tax tables, shipping tables, volume discount tables etc. If the `table()` function does not find an exact match between the key value and the key field, it will pick the record in the target database where the key field is closest to, but not

greater than, than the key value. For example, suppose the key field contains the values 5, 25, 100, 250 and 1000. If the key value is 47, the `table()` function will match with the record containing 25 in the key field. If the key value is 4700, the `table()` function will match with the record containing 1000 in the key field. If the key value is 4, there is no match, because there is no value in the key field less than 4. In this case the default value will be used.

Suppose you have a database called Shipping Rates that contains the fields and values shown here.

Weight	Zone1	Zone2	Zone3
0	2.50	4.00	5.00
50	2.35	3.80	4.70
100	2.25	3.60	4.50
250	2.12	3.40	4.25
500	2.03	3.00	4.05
1000	1.94	2.85	3.85
2000	1.86	2.70	3.70

The `table()` function interprets this table like this: From 0-49 pounds in Zone 1, the rate is \$2.50 per pound. From 50-99 pounds the rate is \$2.35/pound. From 100-249 the rate is \$2.25 per pound, and so on. Items 2,000 pounds and over are shipped for \$1.86 per pound. The other zones are similar.

The procedure below calculates the shipping charges for a package.

```
local PackageWeight, DestinationZone, ShippingCharge
PackageWeight="" DestinationZone=""
getttext "Package weight:", PackageWeight
PackageWeight=val(PackageWeight)
if PackageWeight<0
    message "Sorry, anti-gravity option not available until 3rd quarter."
    stop
endif
getttext "Zone Number (1-3)", DestinationZone
if length(DestinationZone)≠1 or DestinationZone<"1" or DestinationZone>"3"
    message "Zone must be from 1 to 3"
    stop
endif
ShippingCharge=PackageWeight*
    table("Shipping Rates", Weight, PackageWeight,
        "Zone"+DestinationZone, 0, 0)
message "Shipping charge is: "+pattern(ShippingCharge, "$#, .##")
```

Notice that this example actually calculates the name of the data field on the fly: either Zone1, Zone2, or Zone3. The data field is still a single field (remember, only one item can be transferred at a time) but we are using a formula to calculate what the name of that field is.

In a real database you probably would not ask the user to enter the zone, but would have another database that would relate zones to zip codes. Here's a simple Zone Chart database that divides the entire USA into three zones based on the first three digits of the zip code.

Zip3	Zone
000	1
300	2
700	3
99:	0

The last value in this table, **99:**, is the smallest value that is greater than the last legal zip code (999) according to the ASCII character order. This record can help catch illegal zip codes. For instance, **ABC** is greater than **99:**, so the Zone will be 0 for this illegal zip code.

The assignment below will turn a regular zip code (Zip) into a zone number according to the Zone Chart database.

```
DestinationZone=table("Zone Chart",Zip3,Zip[1,3],Zone,0,0)
```

This assignment can easily be plugged into the previous example to calculate the shipping charges given the weight and zip code.

Clairvoyance and Lookups

When data is transferred between two databases with a lookup, there is usually one field in the current database that contains the key value. For example, when we look up a company name from the Customer database, we usually get the name from the Company field in the current database. For the purposes of this discussion we'll call this field in the current database the key value field. This is different from the key field, which is in the target database. However, both of these fields contain the same data: the same customer names, or the same part descriptions, the same account numbers, etc.

Since the key value field and the key field contain the same data, it makes sense to link them with Panorama's Clairvoyance[®] feature. This allows you to type in the first few letters of a customer name (or a part description, or an account number, etc.) and then let Clairvoyance complete the entry from the data in the key field in the target database. When it is used this way, Clairvoyance is almost like a pre-lookup. The beauty of this scheme is that when Clairvoyance completes the entry for you, you know that the key value is entered 100% correctly and that the lookup will work perfectly.

To set up Clairvoyance between fields in different databases you'll need to use the design sheet. Open the design sheet for the current database, then click on the name of the field that contains the key value. Now choose Clairvoyance Link from the Special menu. A dialog with a list of databases will appear. Choose the target database. Now a list of fields in the target database will appear. Choose the key field. Press the **OK** button, then use the New Generation tool to update the database itself. Your cross-database Clairvoyance link is now set up. (Of course, this link will only work if both databases are open.)

The SpeedCopy Statement (Multiple Assignments in One Statement)

Some applications require you to transfer many fields from one database to another. Normally this requires a separate assignment for each field you want to move. This isn't so bad for a few fields, but if you need to transfer, say, 20 fields it gets tedious and slow. The **speedcopy** statement can transfer many fields at once, but only if the fields to be copied in the two databases match exactly. The fields to be copied must appear in exactly the same order in both databases, and the fields must have the same data types. With all these restrictions, you may be surprised to find out that the fields do not have to have the same names!

Here's how **speedcopy** works. Before you use **speedcopy**, you must perform an assignment with a **lookup(** function (actually, any single record transfer function will do: **lookuplast(**, **lookupselected(**, etc.). The **lookup(** function locates the record containing the information to be copied.

The **speedcopy** statement has three parameters.

```
speedcopy FirstAssignField,LastAssignField,FirstTargetField
```

The first two parameters are fields in the current database. The last parameter is a field in the target database. All of these field names should be surrounded by quotes (for example "Name", not Name). **Speedcopy** starts by converting these field names into field numbers. For example, if a field would be the third column in the data sheet, it is field #3.

Once `speedcopy` has converted the field names into numbers, it starts copying data. Suppose the `FirstAssignField` was field number 3, and the `FirstTargetField` was field number 8. `Speedcopy` will start by copying field #8 in the target database into field #3 in the current database. Then it will copy field #9 in the target database into field #4 in the current database. It will continue copying fields until it has copied something into the `LastAssignField`.

To show a specific example, suppose we have two databases, Organizer and Customers, with the fields listed below:

#	Organizer	Customers
1	Name	Company
2	Title	Address
3	Company	City
4	Address	State
5	City	ZipCode
6	State	Phone#
7	Zip	Fax#
8	Phone	Cust#

The procedure below will quickly copy the Address, City, State, Zip and Phone fields from the Customers database to the Organizer database.

```
Address=lookup("Customers",Company,Company,Address,"",0)
if Address!=" "
    speedcopy "City","Phone","City"
endif
```

Let's take a close look at how this procedure works. The first line attempts to lookup the Address from the Customer database. If this lookup fails, the procedure is finished. However, if the lookup succeeds the procedure continues with the `speedcopy` statement.

The first parameter of the `speedcopy` statement is City, which is field #5 in the current database (Organizer). The second parameter is Phone, which is field #8 in the current database. The final parameter is City, which is field #3 in the target database (Customers).

In this example `speedcopy` will copy 4 fields from Customers into Organizer. These four data moves are:

City into City(Customers field #3 into Organizer field #5)

State into State(Customers field #4 into Organizer field #6)

ZipCode into Zip(Customers field #5 into Organizer field #7)

Phone# into Phone(Customers field #6 into Organizer field #8)

As `speedcopy` moves data from one database to another, it doesn't make any kind of checks on the data. If the fields aren't really in the same order, `speedcopy` will cheerfully copy them in the wrong order. Even worse, if you try to copy a numeric field into a text field or a text field into a numeric field, `speedcopy` will not object, but will speedily turn your current database into swiss cheese. The moral of the story is to use the `speedcopy` statement very carefully. Like any sharp instrument you want to make sure it is pointed in the right direction before you use it.

Multiple Record Transfer Functions

The transfer functions described below locate every matching record in the target database, and return all the data field values in the records that match. The data field values are strung together as a single piece of text, with the separator character (or characters) in between each item. If the separator is a single character, the result is a text array.

lookupall(target database,key field,key data,data field,separator)

The **lookupall()** function builds a text array containing one item for every record in the target database where the data in the key field matches the key value. Each item in the text array contains the value extracted from the data field for that record. If the data field is a numeric or date field, it is converted to text using the default patterns for that field.

Here is an example that looks up all the checks written to a certain person or company. The checks are displayed with a comma in between each check number.

```
local CheckTo,Checks
CheckTo=""
gettext "List checks written to:",CheckTo
Checks=lookupall("Checkbook",Payee,CheckTo,«Check#»,",")
if Checks=""
    message "No checks written to "+CheckTo
else
    message "Checks written to "+CheckTo+": "+Checks
endif
```

The **lookupall()** function will often return a lot of duplicate data. Since the result is an array, you can use the **arraydeduplicate** statement to sort and eliminate the duplicates. This example produces a sorted list of customers in Arizona.

```
global theCustomers
theCustomers=lookupall("Invoices",State,"AZ",Company,¶)
arraydeduplicate theCustomers,theCustomers,¶
```

The **lookupall()** function is especially useful for displaying or printing lists of items, and for other user interface elements like lists or pop-up menus. The **lookupall()** function can be used directly in auto-wrap text or a Text Display SuperObject to display a list, or as part of the data formula for a Super Matrix object.

lookupcalendar(target database,key field,key data,data field,separator)

The **lookupcalendar()** function is identical to the **lookupall()** function except for the way that the key value and the key field are compared. The **lookupcalendar()** function requires that the key field be a field that contains reminders, and the key value be a date. (To refresh your memory, a reminder is a special data type that holds scheduling information.) The **lookupcalendar()** function will locate all records where the reminder and the date match. For example, if the date specified is **5/23/95**, **lookupcalendar()** will match for reminders that are set for 5/23/95, or are set for 5/23 of every year (annually), or are set for every Tuesday, etc.

As you might guess, this function is very handy for calendars. Using this function you can display all the reminders for a specific date.

lookupptime(target database,key field,key data,pattern,separator)

Lookupptime is short for lookup reminder time. The **lookupptime()** function is similar to the **lookupcalendar()** function, but it returns the actual time of the reminders instead of information in a separate data field. Like **lookupcalendar()**, the key field must be a field that contains reminders, and the key value must be a date. The pattern parameter tells the function how to format the time into text, for example **"hh:mm am/pm"**. You may use any pattern supported by the **timepattern()** function.

The example below will fill the global variable Agenda with the items on today's schedule.

```
global Agenda
local todayTimes, todayMessages
todayTimes=lookupptime("Reminders", When, today(), "hh:mm am/pm", ¶)
todayMessages=lookupcalendar("Reminders", When, today(), Message, ¶)
arrayfilter todayTimes, Agenda, ¶
  import()+" - "+array(todayMessages, seq()-1, ¶)
```

The list of items in Agenda will be formatted something like this:

```
7:00 am - Breakfast meeting with Bob
9:25 am - Make sure Williams got our quote
1:30 pm - Late lunch with Jennings group
3:45 pm - get prepped for staff meeting
4:00 pm - Weekly staff meeting
5:30 pm - Don't forget flowers for Pat
```

This list can easily be displayed with an auto-wrap text object, a Text Display SuperObject, or a List SuperObject.

After a Lookup...Modifying the Original Data

There is no specific function or statement that allows you to modify data you have looked up. However, it's not difficult to write a procedure that does this. The basic principle is to use the find statement to locate the data, then modify the data.

The example below looks up a price from an Inventory database, then goes to the Inventory database, locates the part with the find statement, decreases the quantity on hand, then goes back to the original window.

```
local originalWindow, theItem, theQty
PriceΩ=lookup("Inventory", Part, ItemΩ, Price, 0, 0)
if PriceΩ=0
  stop
endif
AmountΩ=QtyΩ*PriceΩ
GrandTotal=sum(AmountΩ)
originalWindow=info("windowname")
theItem=ItemΩ
theQty=QtyΩ
window "Inventory"
if info("records")>info("selected")
  selectall
endif
find Part=theItem
QtyOnHand=QtyOnHand-theQty
window originalWindow
```

(This example is actually a bit unrealistic because the normally you would not want the inventory to be updated immediately as each price was looked up. Instead you would probably wait until the entire invoice was complete and post all the inventory changes at one time. Nevertheless, the basic technique is the same.)

The search technique is a bit awkward to use for modifying data that was retrieved by a `lookupall()` function (or any multiple record transfer function). For example, suppose you want to modify the third item retrieved by the `lookupall()` function. You could use the `find` statement in a loop to find the third match in the target database, but Panorama has a special statement that helps perform this job: the `allindex` statement. This statement has two parameters:

```
allindex item,transferformula
```

The `item` parameter must be a variable. Before you use the `allindex` statement you must set this variable to the number of the item you want to locate: 1, 2, 3, etc. Afterwards the variable will contain the line number in the target database that corresponds to this item, or zero if there is no such item.

The second parameter, `transferformula`, should contain the multiple record transfer function (`lookupall()`, `lookupcalendar()`, etc.).

The previous section showed an example that displayed an agenda for a particular day. For this example we'll assume that this agenda has been loaded into a List SuperObject™, and that this SuperObject has been set up to trigger the procedure below whenever the user clicks on an item. This procedure will find out what item the user selected, then use the `allindex` and `find` statements to locate the original record in the Reminders database for that item.

```
local AgendaItem
AgendaItem=1
superobject "AgendaList","FindSelected",AgendaItem
if AgendaItem=0
    stop /* nothing clicked! */
endif
allindex AgendaItem,lookupcalendar("Reminders",When,today(),Message,¶)
window "Reminders"
if info("selected") < info("records") selectall endif
find seq()=AgendaItem
/* now we can modify the original data the mouse was pointing to */
editreminder When,Message
```

The `allindex` statement only works if you display the data directly from the `lookupall()` function (or other multiple record lookup.) If the data is filtered, sorted and/or de-duplicated the `allindex` statement will not be able to correctly locate the original data.

Using Lookups for Display/Printing

So far, all our examples have used lookups and other transfer functions to actually transfer data from one database to another via transfer functions. However, since these are functions, they can be used in a formula anywhere. For example, you can use a lookup in an auto-wrap text object or in a Text Display or Text Editor SuperObject.

Let's take an invoice file to show some examples of how lookups can be used for display. If all of your customers are listed in a customer database, you can display the address on the invoice without actually having fields in the database for the address. Use an auto-wrap text object or Text Display SuperObject with a lookup formula to display the address. One advantage of this approach is that if you change the address of the company in the customer database, all the invoices will automatically show the new address. A possible disadvantage of this approach is that there is no way to enter the address of a company that is not in the customer database. You must enter the company in the customer database first, and then create the invoice.

Another application on an invoice would be to display a list of all the previous invoices for this customer. To do this you would use the `lookupall()` function. As you flip from invoice to invoice the new list is automatically displayed. You might want to use more than one auto-wrap text object or Text Display object; perhaps one to display the invoice numbers, one to display the invoice dates, and one to display the invoice amounts. If these are placed next to each other, they will line up in a neat columnar display.

One place where you probably don't want to simply display the result of a lookup is prices. We've already shown examples where a price is transferred from a price list database into the invoice database. You could simply display the price instead of storing it in the invoice...you can even perform calculations on the price as part of the display formula. But what if the price list changes? You probably don't want your old invoices to change every time a price changes—these old invoices are a historical record of the transactions that actually took place. By transferring the price into the invoice database you are “freezing” it and isolating it from later changes in the price list database.

The point of this section is: Don't always assume that you must transfer data from one database to another to link them. Look at each situation carefully. Do I need to transfer the data to isolate it from further changes? Do I want the data to continue to be linked and to reflect changes in the original database? How much additional processing do I need to do to this data after it has been transferred? Don't always do it the same way, but pick the best solution for each individual situation.

Using ArrayBuild to Transfer Data Between Files

The `arraybuild` statement can scan any open database and extract data from it. Does that sound familiar? The `arraybuild` can act as a super duper lookup with several advantages. The table below summarizes the differences between using regular lookup functions and the `arraybuild` statement for retrieving data from another database:

	Lookup Functions	ArrayBuild Statement
Locating Data	Only 1 key field	Can combine multiple fields
Match Criteria	Exact match required	Any comparison operator (=, <, >, contains, soundslike, match, etc.)
Retrieving Data	Only 1 data field	Can combine multiple fields
Use in Procedure	Yes	Yes
Use in Form	Yes	No

The primary disadvantage of the `arraybuild` statement is that it is not a function, so it cannot be used as part of a formula. The `arraybuild` statement can only be used as part of a procedure.

Let's start by looking at how the `arraybuild` statement can be used to simulate various transfer functions. Here's an example from earlier in this chapter that uses the `lookup()` function.

```
gettext "What company name?",Company
Address=lookup("Customers",Company,Company,Address,"",0)
City=lookup("Customers",Company,Company,City,"",0)
State=lookup("Customers",Company,Company,State,"",0)
Zip=lookup("Customers",Company,Company,Zip,"",0)
```

Here's another procedure that does the same thing but using the `arraybuild` statement.

```
local theCompany,TransferArray
gettext "What company name?",Company
theCompany=Company
arraybuild TransferArray,1,"Customers",
  ?(theCompany=Company,Address+--+City+--+State+--+Zip,"")
if TransferArray=""
  stop
endif
Address=array(array(TransferArray,1,1),1,-)
City=array(array(TransferArray,1,2),2,-)
State=array(array(TransferArray,1,3),3,-)
Zip=array(array(TransferArray,1,4),4,-)
```

Ok, ok...I know you're not impressed (yet!). This example is longer and more obscure looking than the first one, and it does the same thing. But now take a look at a procedure that is impossible with the `lookup()` function. This procedure looks up a person by their last name and first initial (separate fields in the Customers database).

```
local theFirst,theLast,TransferArray
theFirst=upper(FirstName) theLast=upper(LastName)
arraybuild TransferArray,¶,"Customers",
    ?(theFirst[1,1]=upper(FirstName[1,1]) and theLast=upper(LastName),
    Address+~+City+~+State+~+Zip,"")
if TransferArray=""
    stop
endif
Address=array(array(TransferArray,1,¶),1,~)
City=array(array(TransferArray,1,¶),2,~)
State=array(array(TransferArray,1,¶),3,~)
Zip=array(array(TransferArray,1,¶),4,~)
```

How does this procedure do its job? First, it transfers the data it wants to look for into temporary variables (`theFirst` and `theLast`). In this example, the data is also converted to all upper case, so that it doesn't matter if the names are upper or lower case in the two different databases.

Now the `arraybuild` statement gets to work. The `?` function uses a formula to identify people by their first initial (`theFirst[1,1]=upper(FirstName[1,1])`) and also by their last name (`theLast=upper(LastName)`). When it encounters a record that matches the formula, it copies the Address, City, State, and Zip into the `TransferArray` variable. These four fields are separated by tabs within the array (remember, the `~` character is a tab). For example, the `TransferArray` might look like this:

124 W. Olive St	San Jose	CA	95134
2347 N. Riverside	Cambridge	MA	02139
687 E. Dorothy Lane	Bothell	WA	98011
5672 Lakewood Drive	Salinas	CA	93908

In this hypothetical example the `arraybuild` statement has located four records in the target database that match the formula. Which one to choose? The procedure could choose the first match, the last match, or it could display the entire list to the user and ask them to make the choice. The procedure we have created always uses the first matching record. The inner `array()` function, `array(TransferArray,1,¶)`, locates the first record that matched from the target database. The outer `array()` function pulls out the individual items: Address, City, etc.

If you wanted to find the last record that matched you would rewrite the end of this procedure like this:

```
...
...
if TransferArray=""
    stop
endif
local LastMatch
LastMatch=arraysize(TransferArray,¶)
Address=array(array(TransferArray,LastMatch,¶),1,~)
City=array(array(TransferArray,LastMatch,¶),2,~)
State=array(array(TransferArray,LastMatch,¶),3,~)
Zip=array(array(TransferArray,LastMatch,¶),4,~)
```

For some applications the best option may be to let the user choose the appropriate record if there are duplicates. To illustrate this, we'll assume that we have created a form called `Select` that contains a List SuperObject named `Choices`. The List SuperObject has been set up to display whatever list is contained in the global variable named `ChoiceList`. The main procedure looks like this:

```
global TransferArray
local theFirst,theLast,HowMany
theFirst=upper(FirstName) theLast=upper(LastName)
arraybuild TransferArray,¶,"Customers",
  ?(theFirst[1,1]=upper(FirstName[1,1]) and theLast=upper(LastName),
  FirstName+" "+LastName+--+Address+--+City+--+State+--+Zip,"")
if TransferArray=""
  stop
endif
HowMany=arraysize(TransferArray,¶)
if HowMany=1
  Address=array(array(TransferArray,1,¶),2,-)
  City=array(array(TransferArray,1,¶),3,-)
  State=array(array(TransferArray,1,¶),4,-)
  Zip=array(array(TransferArray,1,¶),5,-)
else
  /* duplicate records, so let user pick from list */
  arrayfilter TransferArray,ChoiceList,¶,
    array(import(),1,-)
  opendialog "Select"
endif
```

If `TransferArray` contains only one record, the procedure simply fills in the `Address`, `City`, etc. But if `TransferArray` contains multiple records, the procedure builds a new array that contains the names of the people it found, and then opens the dialog to display the list.

When the user clicks on one of the names in the List SuperObject we'll need another procedure that actually fills in the `Address`, `City`, etc. Here is that procedure:

```
local UserChoice
UserChoice=1
superobject "Choices","FindSelected",UserChoice
if UserChoice=0 stop endif ; nothing clicked!
closewindow ; close the dialog
UserChoice=UserChoice
Address=array(array(TransferArray,UserChoice,¶),2,-)
City=array(array(TransferArray,UserChoice,¶),3,-)
State=array(array(TransferArray,UserChoice,¶),4,-)
Zip=array(array(TransferArray,UserChoice,¶),5,-)
```

If you've been studying these examples carefully you have probably noticed that in these three examples we've used the `arraybuild` statement in a manner similar to the `lookup()`, `lookuplast()` and `lookupall()` functions—but with much more flexible parameters. The possibilities for linking databases with the `arraybuild` statement are almost limitless.

Posting Data to Other Databases

So far, all of our examples have pulled data from other database files into the current record of the currently open database. What if you need to go the other way? What if you have information in the current database that needs to be moved somewhere else? This is often called “posting” data, and is essentially performing a lookup in reverse. Panorama has two statements for posting data — `post` and `postadjust`.

The Post Statement

The `post` statement assigns values to one or more fields in another database.

```
post mode,database,keyfield,keyvalue,datafield,datavalue,df2,dv2,df3,dv3 ...
```

The `mode` parameter controls how the `post` statement decides which record will be updated in the target database. There are three modes available:

Mode	Description
"update"	When this mode is used, the <code>post</code> statement will search for a record where the data in the <code>keyfield</code> matches the <code>keyvalue</code> you have supplied. When this record is found, the specified datafields are updated. If a record with matching data does not exist an error will be generated (this error can be trapped with the <code>if error</code> statement).
"add"	When this mode is used, the <code>post</code> statement will not search for a matching record, in fact the <code>keyfield</code> and <code>keyvalue</code> parameters are ignored. The statement simply adds a brand new record and updates the specified data fields.
"updateadd"	When this mode is used, the <code>post</code> statement will search for a record where the data in the <code>keyfield</code> matches the <code>keyvalue</code> you have supplied. If this record is found, the specified datafields are updated. If a record with matching data does not exist a new record is added and the specified datafields are updated.

The next five parameters are very similar to the parameters for the `lookup()` function.

The `database` parameter is the name of the target database you want to post data to. This database must already be open.

The `keyfield` parameter is the name of the field in the target database you want to search (`update` and `updateadd` modes only).

The `keyvalue` parameter is a formula that specifies the value you want to search for in the `keyfield` field.

The `datafield` parameter is the name of the field that will be modified. The `datavalue` is the new value to be placed in this field. You may specify additional pairs of fields/values to modify in the target database, in fact, there is no limit to the number of field/value pairs you can include as parameters to this statement.

This example assigns the value "714" to the AreaCode field in the "My Contacts" database and it will assign "555-1212" to the Phone field and "4" to the Extension field. It will look for a matching customer name and create a new record if it can't find one.

```
post "updateadd","My Contacts","CustomerName",CustomerName,
    "AreaCode","714",
    "Phone","555-1212",
    "Extension","4"
```

The PostAdjust Statement

The `postadjust` statement adjusts the value of a numeric field in another database, by adding (or subtracting) a number. For example, you could use this to subtract from the quantity on hand field of an inventory database as an invoice is processed.

```
postadjust database,keyfield,keyvalue,datafield,deltavalue,minimum,maximum
```

Unlike the `post` statement, the `postadjust` statement always runs in `update` mode. If the `postadjust` statement cannot find the record specified by the `keyfield` and `keyvalue` parameters it will stop and generate an error (you can trap this error with the `if error` statement).

The first four parameters are very similar to the parameters for the `lookup()` function.

The `database` parameter is the name of the target database you want to post data to. This database must already be open.

The `keyfield` parameter is the name of the field in the target database you want to search.

The `keyvalue` parameter is a formula that specifies the value you want to search for in the `keyfield` field.

The `datafield` parameter is the name of the field that will be modified. This must be a numeric field.

The `deltavalue` is the adjustment you want to make to the `datafield`. This should be a positive or negative number. For example, a `deltavalue` of `-6` means “subtract six from the `datafield`.”

The `minimum` parameter allows you to specify a minimum value for the `datafield`. If the adjusted value of this field would be below this value then the adjustment is not made and an error is generated. For example when adjusting an inventory database you would typically set the minimum to zero to prevent the quantity on hand from becoming negative. The error message will look like the one shown below. This error indicates that you tried to subtract 8 items from 5, which would result in `-3`, which is below the minimum.

```
POSTADJUST failed because adjusted value would be below minimum.
Database Value:5 Delta:8
```

If you trap the error with the `if error` statement you can extract these values with the `info("error")` function. An example of this is shown below.

If you don't want to have any minimum value at all, set the `minimum` parameter to `-1`.

The `maximum` parameter allows you to specify a maximum value for the `datafield`. If the adjusted value of this field would be above this value then the adjustment is not made and an error is generated. For example if you were adjusting a class size you might set the maximum to the number of seats in the classroom. The error message will look like the one shown below. This error indicates that you tried to add 3 items to 285, which would result in 31, which is above the maximum of 30.

```
POSTADJUST failed because adjusted value would be above maximum.
Database Value:28 Delta:3
```

If you trap the error with the `if error` statement you can extract these values with the `info("error")` function.

If you don't want to have any maximum value at all, set the `maximum` parameter to `0`.

The example below shows how to link an inventory and invoice database so that the inventory quantities on hand are automatically adjusted as invoices are shipped, and back orders are marked as necessary.

The inventory database includes at least two fields for each product.

Field	Description
SKU	Identification code for each product
OnHand	Current quantity on hand

The invoice database includes at least these four line item fields for each item purchased (presumably it would also include fields with pricing information).

Field	Description
SKU1, SKU2, ...	Identification code for product
Qty1, Qty2, ...	Quantity customer desires to purchase
Shipped1, Shipped2, ...	Total quantity shipped to customer
ShipNow1, ShipNow2, ...	Quantity to ship now

The procedure below prepares an invoice for shipping. It scans each line item and determines if it needs to be shipped. If one or more items remain unshipped, it checks the inventory database. If there is insufficient inventory on hand to complete the order, it grabs whatever is available.

```

local itemNumber,itemSKU,itemQty,itemShipped,itemNeeded,itemNow,itemAvailable
itemNumber=1
loop
  itemSKU=grabdata("Invoice","SKU"+str(itemNumber))
  if error
    stoploopif true() /* loop stops when line item field does not exist */
  endif
  if itemSKU<>" /* ignore blank line items */
    itemQty=grabdata("Invoice","Qty"+str(itemNumber))
    itemShipped=grabdata("Invoice","Shipped"+str(itemNumber))
    if itemQty<itemShipped/* is everything shipped yet? */
      itemNeeded=itemQty-itemShipped
      itemNow=itemNeeded
      postadjust "Inventory","SKU",itemSKU,"OnHand",-itemNeeded,0,0
      if error
        itemNow=0
        itemAvailable=val(tagdata(info("error"),"Database Value:"," ",1))
        if itemAvailable>0
          /* not enough to fill order, just ask for what we can get */
          itemNow=itemAvailable
          postadjust "Inventory","SKU",itemSKU,"OnHand",-itemNow,0,0
        endif
      endif
    endif
    set "Shipped"+str(itemNumber),itemShipped+itemNow
    set "ShipNow"+str(itemNumber),itemNow
  endif
  itemNumber=itemNumber+1
while forever

```

When the procedure is done the inventory has been updated and the invoice will contain the quantities to ship now, if any.

Sorting

To sort the database takes two steps. First the procedure must select the field to sort by (see “[Moving Left and Right](#)” on page 515). Then the `sortup` or `sortdown` statement is used to sort the database. The `sortup` statement sorts the database in ascending order — A’s at the top and Z’s at the bottom. The `sortdown` statement does the reverse, Z’s go at the top and A’s at the bottom. This example sorts an address book by last name.

```
field "Last Name"
sortup
```

To sort by two or more fields you must use the `sortupwithin` statement. This statement leaves the data in the original field in order but re-arranges the records within each value. For example the procedure below will sort an address book by state, and then by city within each state.

```
field State
sortup
field City
sortupwithin
```

Note: Because Panorama uses what is called a “stable sort algorithm” there is another way to sort multiple fields. Instead of using `sortupwithin` you can sort the fields in reverse order, like this.

```
field City
sortup
field State
sortup
```

Just like the previous example, this procedure will sort the address by city within state. There really isn’t any advantage to using this technique, but it is available.

To sort the database by the color of the data in a field use the `sortbycolor` statement. See “[Sorting By Color](#)” on page 327 of the *Panorama Handbook* to learn more about sorting by color.

Reducing Screen “Flashing”

When a database is sorted more than once in a row the window will redisplay over and over again. This is annoying and wastes time, also. You can eliminate this “flashing” with the `noshow` and `endnoshow` statements. This example shows a revised version of our procedure to sort an address book by city and state.

```
noshow
  field City
  sortup
  field State
  sortup
  showpage
endnoshow
```

This revised procedure will only redisplay the window once (because of the `showpage` statement). To learn more about the `noshow` and `endnoshow` statements see “[Suppressing Display of Text and Graphics](#)” on page 307.

Making Sorts Even Faster

Panorama sorts even large databases very quickly. However, the `noundo` statement can make it sort even faster. This statement disables Panorama’s undo feature. Since the sort doesn’t need to worry about undo it can run slightly faster.

```
noundo
field Company
sortup
```

Locating Information

Panorama has two ways of locating information — **finding** and **selecting** (see “[Finding vs. Selecting](#)” on page 331). In a procedure you find information with the **find** statement and select information with the **select** statement.

Finding Information

The **find** statement searches the database, starting from the top. This statement has one parameter, a formula. Starting from the top of the selected records, Panorama scans down the database until it finds a record that makes this formula true (see “[True/False Formulas](#)” on page 124). For example, this procedure will scan down the database until it finds a record where the **Park** field contains **Everglades**.

```
find Park contains "Everglades"
```

Notice that the active field stays the same (**City**) even though the formula searches the **Park** field.




Park	Address	City	Sta	Zip	Phone	Fee	URL
Assateague Island National Seashore	7206 National Highway 17	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/hatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov/gettysburg

After the **find** statement you can check to see if Panorama actually found anything with the **info("found")** function. Here is a procedure that uses this statement and function to locate a park and, if found, make it bold (see “[Data Style and Color](#)” on page 588).

```
local choice,wasField
choice=""
gettext "Which park?",choice
find Park contains choice
if info("found")
    wasField=info("fieldname")
    field Park
    Style "cell bold"
    field (wasField)
else
    message "Sorry, park not found."
endif
```

When you run this procedure it starts by asking you what park you want to highlight (see “[Basic Text Entry Dialogs](#)” on page 480).



Which park?

grand canyon

Stop OK

If the `find` statement locates the requested information it marks the park name in bold.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.r
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyo	UT	84717	(435) 834-5322	\$20.00	http://www.r
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.r
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.r
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.r
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.r
Everglades National Park	40001 State R	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.r
Fire Island National Seashore	120 Laurel Str	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.r
Gettysburg National Military Park	97 Taneytown I	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.r
Glacier National Park	P.O. Box 128	West Glacier	MT	59936	(406) 888-7800	\$5.00	http://www.r
Grand Canyon National Park	P.O. Box 129	Grand Canyo	AZ	86023	(520) 638-2631	\$10.00	http://www.r
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.r
Great Basin National Park		Baker	NV	89311	(775) 234-7331	\$0.00	http://www.r

If the requested information is not found then an error message is displayed, but the database is not modified.

A Handy Universal Find Procedure

Here is a handy procedure that will search every field in the database. It can be used in any database without modifications. (Note: A variation of this procedure is in the **Search All Fields** wizard.)

```
local whatfor
whatfor=""
gettext "Search for?",whatfor
find exportline() contains whatfor
```

The secret of this procedure is the `exportline()` function (see "[EXPORTLINE\(\)](#)" on page 5210 of the *Panorama Reference*). This function takes all the fields in a line, converts them to text if necessary, and then appends them together with tabs in between. The procedure uses this handy capability to search all of the fields in the database at once!

Here is a slightly revised version of this procedure that is even cooler. If it finds what you are looking for it automatically moves to the field containing the data it has located.

```
fileglobal whatfor
whatfor=""
gettext "Search for?",whatfor
find exportline() contains whatfor
if info("found")
    field (array(dbinf("fields",""),
        arrayelement(exportline(),search(upper(exportline()),upper(whatfor)),-),1))
else
    beep
endif
```

When you run this procedure it stops and asks you what you want to look for.



Search for?

If that word or phrase exists anywhere in the database, it will find it.

A screenshot of a spreadsheet titled "National Parks" with columns: Park, Address, City, Sta, Zip, Phone Number, Fee, and URL. The row for "West Glacier" is highlighted in blue, with the text "West Glacier" in the City column.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Seashore	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov
Bryce Canyon National Park	P.O. Box 170001	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov
Cape Hatteras National Seashore	Route 1; Box 675	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov
Everglades National Park	40001 State Road 9336	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov
Glacier National Park	P.O. Box 128	West Glacier	MT	59936	(406) 888-7800	\$5.00	http://www.nps.gov
Grand Canyon National Park	P.O. Box 129	Grand Canyon	AZ	86023	(520) 638-2631	\$10.00	http://www.nps.gov

It can find a phone number like 882-4336.

A screenshot of a spreadsheet titled "National Parks" with columns: Park, Address, City, Sta, Zip, Phone Number, Fee, and URL. The row for "Cumberland Island National Seashore" is highlighted in blue, with the phone number "(912) 882-4336" in the Phone Number column.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Seashore	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov
Bryce Canyon National Park	P.O. Box 170001	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov
Cape Hatteras National Seashore	Route 1; Box 675	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov
Everglades National Park	40001 State Road 9336	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov

Or it can even find a numeric value like 10.00.

A screenshot of a spreadsheet titled "National Parks" with columns: Park, Address, City, Sta, Zip, Phone Number, Fee, and URL. The row for "Death Valley National Park" is highlighted in blue, with the fee "\$10.00" in the Fee column.

Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Seashore	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov
Bryce Canyon National Park	P.O. Box 170001	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov
Cape Hatteras National Seashore	Route 1; Box 675	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov
Everglades National Park	40001 State Road 9336	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov

Wherever the data is, this procedure will find it and move right to the spot. See the next section for a “universal find next” procedure to go with this procedure.

Find Next

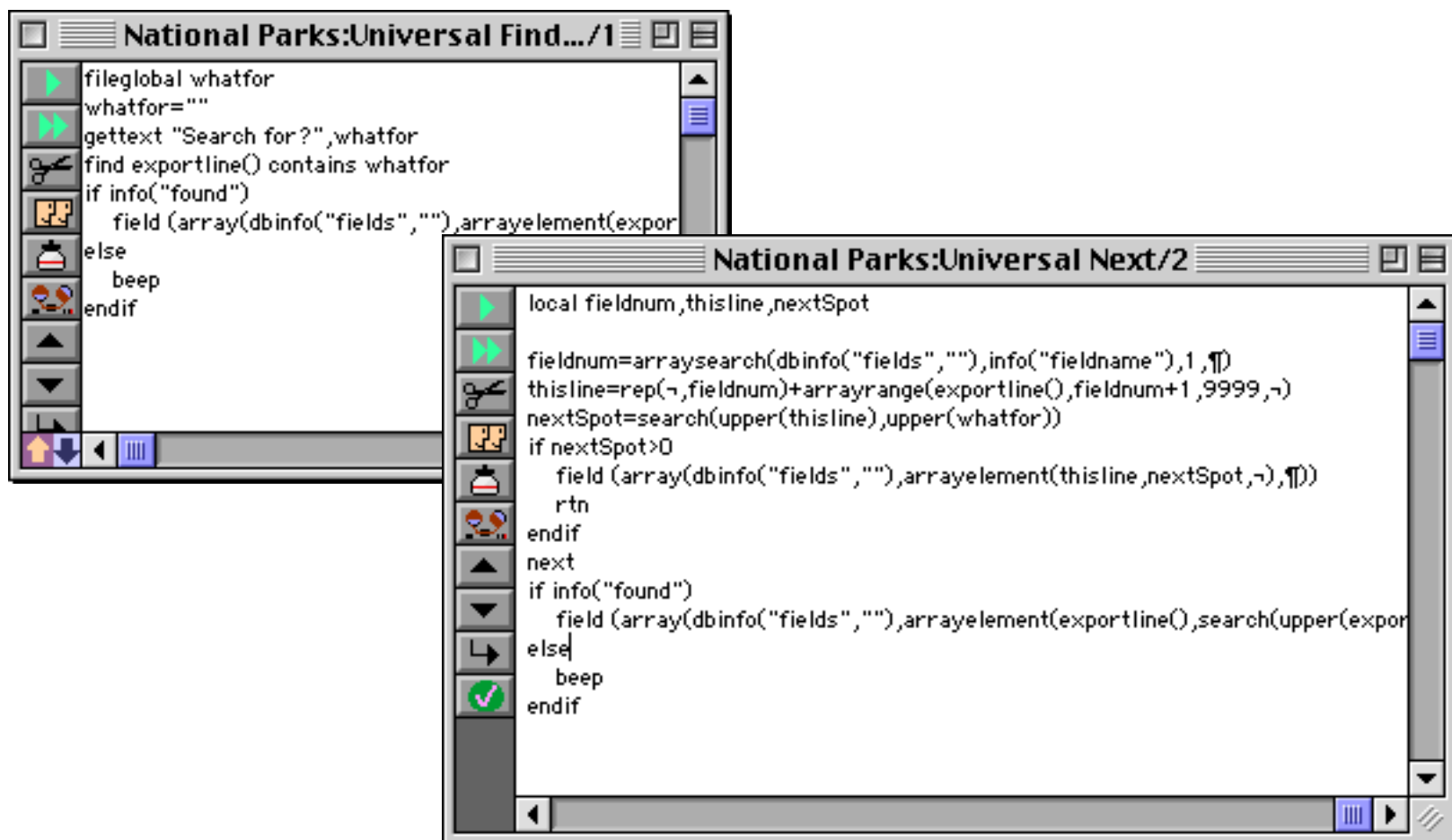
To find the next match use the `next` statement. This is just like the `find` statement except there is no formula...it re-uses the formula supplied with the `find` statement. You can continue to use the `next` statement over and over again until the `info("found")` function tells you there are no more matches. The procedure below uses the `next` statement to find the next occurrence of the word or phrase — either on the same line or on a different line.

```

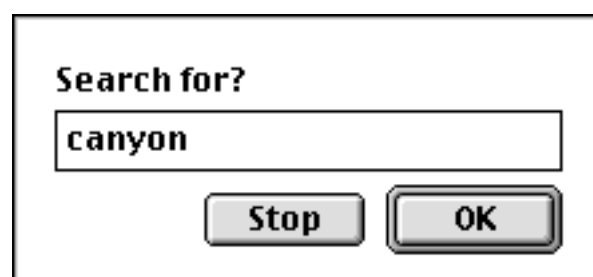
local fieldnum,thisline,nextSpot
fieldnum=arraysearch(dbinfo("fields",""),info("fieldname"),1,1)
thisline=rep(↵,fieldnum)+arrayrange(exportline(),fieldnum+1,9999,↵)
nextSpot=search(upper(thisline),upper(whatfor))
if nextSpot>0
    field (array(dbinfo("fields",""),arrayelement(thisline,nextSpot,↵),1))
    rtn
endif
next
if info("found")
    field (array(dbinfo("fields",""),
        arrayelement(exportline(),search(upper(exportline()),upper(whatfor)),↵),1))
else
    beep
endif

```

To illustrate these universal find procedures we've added them to the [National Parks](#) database.



To demonstrate these procedures we'll start by running `Universal Find` to search for `canyon`.



When the **OK** button is pressed the procedure finds the first occurrence of the word **canyon**.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/capehatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland

Running the **Universal Next** procedure locates the next occurrence of the word **canyon**, in the City column of the same record.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/capehatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland

There's no more occurrences of the word **canyon** in this record, so choosing **Universal Next** again jumps down several lines.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/capehatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov/gettysburg
Glacier National Park	P.O. Box 128	West Glacier	MT	59936	(406) 888-7800	\$5.00	http://www.nps.gov/glacier
Grand Canyon National Park	P.O. Box 129	Grand Canyon	AZ	86023	(520) 638-2631	\$10.00	http://www.nps.gov/grandcanyon
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.nps.gov/grandteton
Great Basin National Park		Baker	NV	89311	(775) 234-7331	\$0.00	http://www.nps.gov/greatbasin

Each time you choose **Universal Next** Panorama will jump to the next occurrence of the word **canyon**, until it finally runs out.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Assateague Island National Seashore	7206 National Highway 1	Berlin	MD	21811	(410) 641-1441	\$5.00	http://www.nps.gov/assateague
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyon	UT	84717	(435) 834-5322	\$20.00	http://www.nps.gov/bryce
Cape Hatteras National Seashore	Route 1, Box 6	Manteo	NC	27954	(252) 473-2111	\$0.00	http://www.nps.gov/capehatteras
Cumberland Island National Seashore	P.O. Box 806	St. Marys	GA	31558	(912) 882-4336	\$4.00	http://www.nps.gov/cumberlandisland
Death Valley National Park	P.O. Box 579	Death Valley	CA	92328	(760) 786-2331	\$10.00	http://www.nps.gov/deathvalley
Denali National Park	P.O. Box 9	Denali Park	AK	99755	(907) 683-2294	\$5.00	http://www.nps.gov/denali
Everglades National Park	40001 State Road 112	Homestead	FL	33034	(305) 242-7700	\$10.00	http://www.nps.gov/everglades
Fire Island National Seashore	120 Laurel Street	Patchogue	NY	11772	(631) 289-4810	\$0.00	http://www.nps.gov/fireisland
Gettysburg National Military Park	97 Taneytown Road	Gettysburg	PA	17325	(717) 334-1123	\$0.00	http://www.nps.gov/gettysburg
Glacier National Park	P.O. Box 128	West Glacier	MT	59936	(406) 888-7800	\$5.00	http://www.nps.gov/glacier
Grand Canyon National Park	P.O. Box 129	Grand Canyon	AZ	86023	(520) 638-2631	\$10.00	http://www.nps.gov/grandcanyon
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.nps.gov/grandteton
Great Basin National Park		Baker	NV	89311	(775) 234-7331	\$0.00	http://www.nps.gov/greatbasin

Let's go back and review the original **Universal Find** procedure for a moment.

```
fileglobal whatfor
whatfor=""
gettext "Search for?",whatfor
find exportline() contains whatfor
if info("found")
    field (array(dbinf("fields",""),
        arrayelement(exportline(),search(upper(exportline()),upper(whatfor)),-),1))
else
    beep
endif
```

The first line of this procedure declares the **fileglobal** variable **whatfor**. It's important that this variable is declared as a **global** or **fileglobal** and not as a **local** variable. Why? Well, remember that Panorama stores the formula used by the **find** statement for use by the **next** statement. In this case the formula is

```
exportline() contains whatfor
```

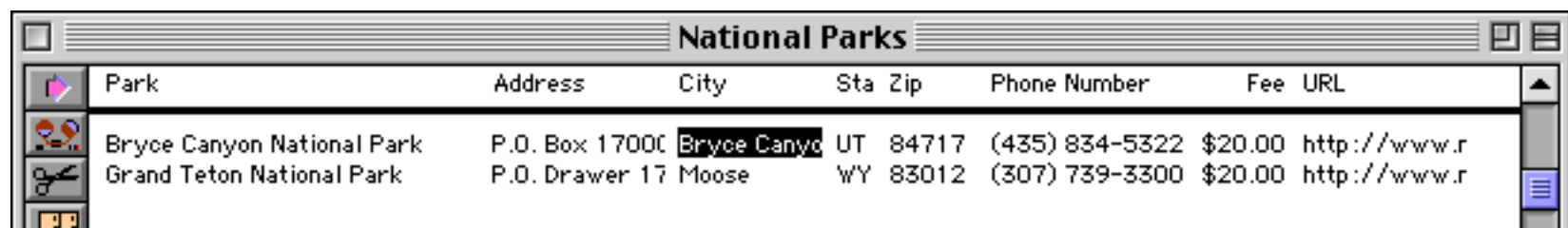
If **whatfor** is a **local** variable, it will cease to exist as soon as the **Universal Find** procedure is finished (see "[The Birth and Death of a Local Variable](#)" on page 249). Because of this when the **next** statement is executed (or if you manually choose **Next** from the Search menu) an error will occur: **field or variable does not exist!** The solution is simply to create **whatfor** as a **global** or **fileglobal** variable so that it will still be hanging around when it becomes time to search for the next occurrence of the word or phrase.

Selecting Information

The **select** statement searches the database, making everything that does not match invisible (see "[Finding vs. Selecting](#)" on page 331 of the *Panorama Handbook*). This statement has one parameter, a formula. Starting from the top of the selected records, Panorama scans down the database looking for records that makes this formula true (see "[True/False Formulas](#)" on page 124). If the formula is not true the record will be made temporarily invisible. For example, this procedure will select all parks where the admission fee is greater than ten dollars.

```
select Fee>10
```

Only two parks in this database fit in this category. All the others are temporarily invisible.

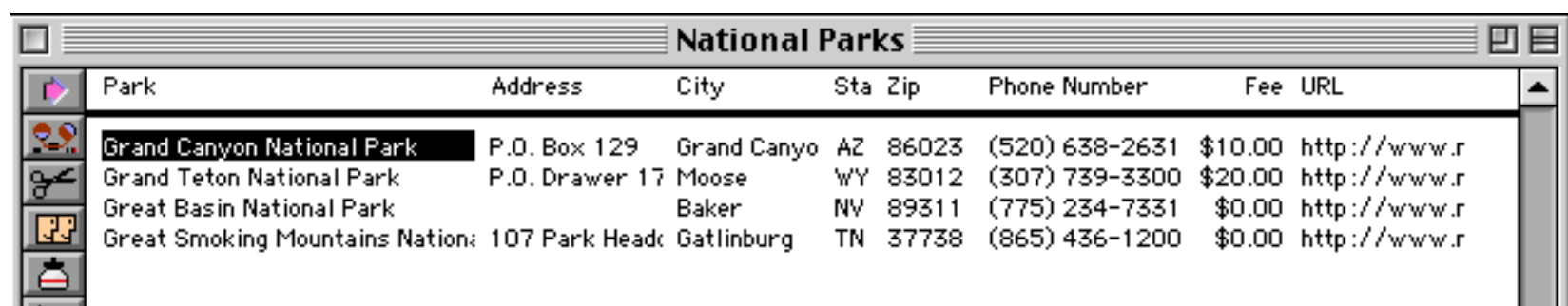


Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Bryce Canyon National Park	P.O. Box 17000	Bryce Canyo	UT	84717	(435) 834-5322	\$20.00	http://www.r
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.r

You can construct as complex a formula as you like, combining different elements together with **and** and **or**.

```
select Park contains "Great" or Park contains "Grand"
```

In this case four records match the criteria.



Park	Address	City	Sta	Zip	Phone Number	Fee	URL
Grand Canyon National Park	P.O. Box 129	Grand Canyo	AZ	86023	(520) 638-2631	\$10.00	http://www.r
Grand Teton National Park	P.O. Drawer 17	Moose	WY	83012	(307) 739-3300	\$20.00	http://www.r
Great Basin National Park		Baker	NV	89311	(775) 234-7331	\$0.00	http://www.r
Great Smoking Mountains Nation:	107 Park Head	Gatlinburg	TN	37738	(865) 436-1200	\$0.00	http://www.r

A procedure can find out how many records are selected with the `info("selected")` function. The `info("records")` function returns the total number of records in the database, both visible and invisible. See “[Database Information](#)” on page 180 for more information on these functions.

When a procedure wants to make sure that all records are selected it should use the `selectall` statement. Here is a simple procedure that checks to see if all records are selected, and if not, selects them.

```
if info("selected")<info("records")
  selectall
endif
```

A procedure can use the `selectadditional` statement to add to the current selection. The `selectwithin` statement can be used to select a subset of the currently selected subset. The `selectreverse` statement swaps the visible and invisible records.

Handling Empty Selections

What if the `select` statement fails to select any records? Eeek! Panorama always requires that at least one record be selected at all times, it never allows every record in a database to be invisible. If none of the records in the database match the formula, Panorama does nothing. It’s as if the `select` never happened. Whatever records were visible before remain visible after. This can be a problem if the following statements are expecting a particular subset of the database to be selected.

Fortunately, Panorama normally handles this situation for you automatically so that your procedures will work correctly. Panorama keeps track of the fact that there should be no records selected, and it will skip any statement that modifies the database, including `formulafill`, `sequence`, `propagate`, `unpropagate`, etc. (basically any statement that corresponds to an item in the Math menu will be skipped). Panorama will continue skipping these statements until it comes to a `selectall` statement or another `select` statement.

Panorama’s automatic statement skipping for empty subsets should work fine for most applications. As a procedure programmer, however, you have the choice of overriding this statement skipping and programming your own solution to the empty subset condition.

To test for an empty subset, use the `info("empty")` function. This example calculates the `InvoiceAge` field only for invoices that have actually been shipped (the `ShipDate` field is not empty) and that are not paid yet. An error message will appear if there are no outstanding invoices.

```
select sizeof(ShipDate)≠0 and Balance>0
if info("empty")
  message "No outstanding invoices!"
else
  field InvoiceAge
  formulafill today()-ShipDate
  selectall
endif
```

Remember, this logic is only necessary if you want to perform some special handling of empty subsets. Normally, Panorama will handle the empty subset just fine on its own by skipping the statements until the selected subset changes.

Selecting Duplicates

The `selectduplicates` statement may be used to locate and select duplicate entries in a database. The statement has one parameter, a formula that determines what fields to check for duplicates. If this parameter is empty text (" ") then the current field is assumed. This statement must be combined with the `sortup` statement. For example, to locate duplicate check number entries within a database you would use this procedure.

```
field "Check Number"
sortup
selectduplicates ""
```

If you supply a formula you can check for duplicates across multiple fields, or using only part of a field, or both, as in the example below. This procedure will check for duplicates in the same zip code, with the same last name and first initial. The formula uses a text funnel to extract the initial from the first name. See “[Taking Strings Apart \(Text Funnels\)](#)” on page 69 if you are not familiar with text funnels.

```
field Zip
sortup
field "Last Name"
sortupwithin
field "First Name"
sortupwithin
selectduplicates Zip+«Last Name»+«First Name»[1,1]
```

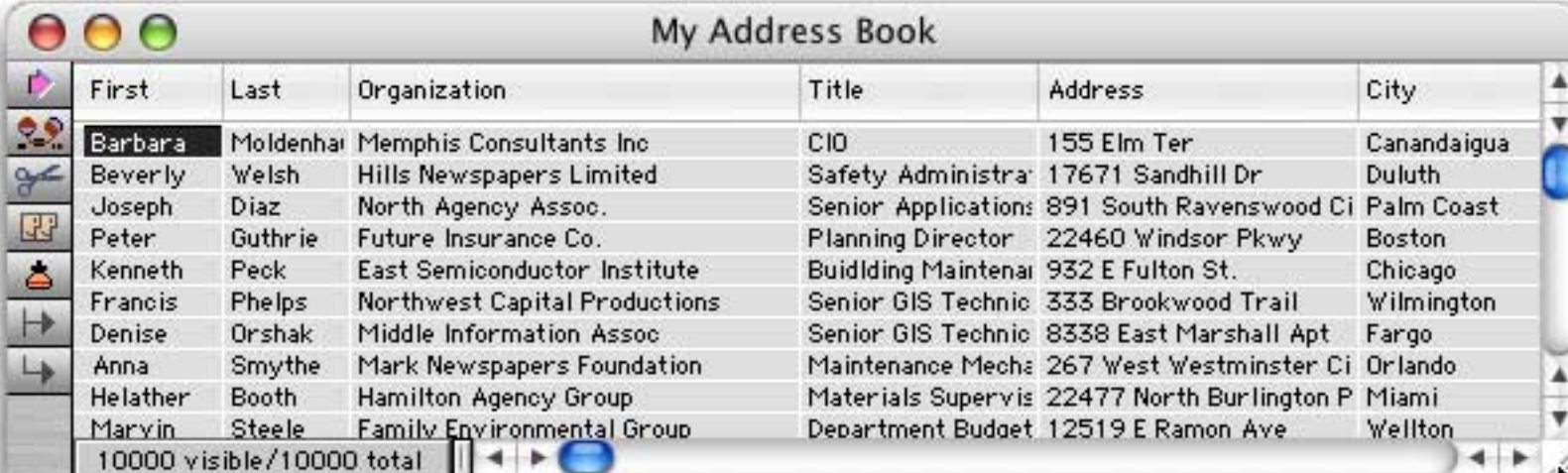
The database must be sorted so that the duplicates you want to find will be consecutively located within the database (see “[Sorting](#)” on page 551).

Live Clairvoyance™

Panorama V introduced a new search method — **Live Clairvoyance™**. Live Clairvoyance allows you to perform "live" searches on any Panorama database. The search results are updated dynamically as you type, allowing you to "hone in" on just the information you are looking for. The search may include multiple fields or even all fields in the database being searched. (If you've used the search box in iTunes you'll find Live Clairvoyance familiar, although the underlying technology is not related.)

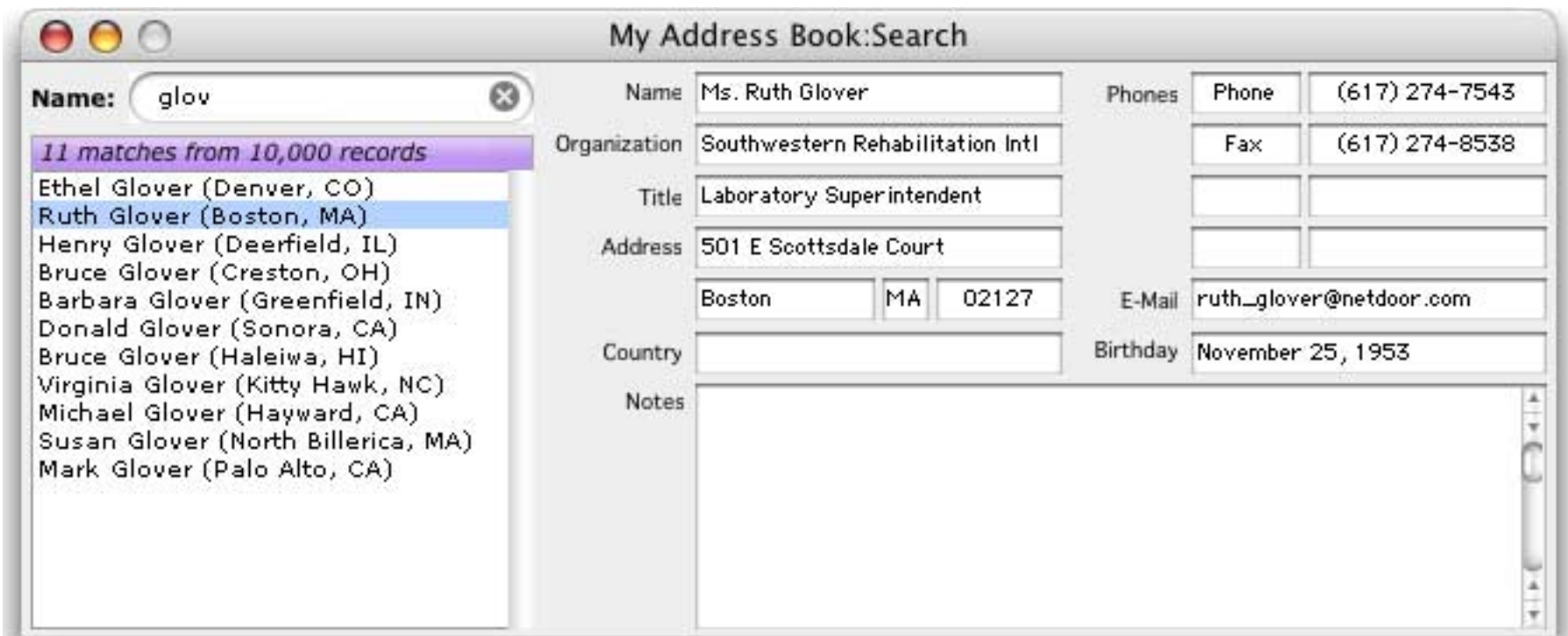
The easiest way to use Live Clairvoyance is with the wizard that comes with Panorama. This wizard performs the neat trick of allowing you to use Live Clairvoyance without doing any programming, and in fact without making any modification at all to your databases. However, for some applications you may want to build Live Clairvoyance into your forms. This allows you to customize it exactly for your needs, and to integrate it with other database operations.

To illustrate how Live Clairvoyance can be used we'll use a database called **My Address Book**. You'll find a copy of this database in the **Guided Tour** submenu of the **Wizard** menu.

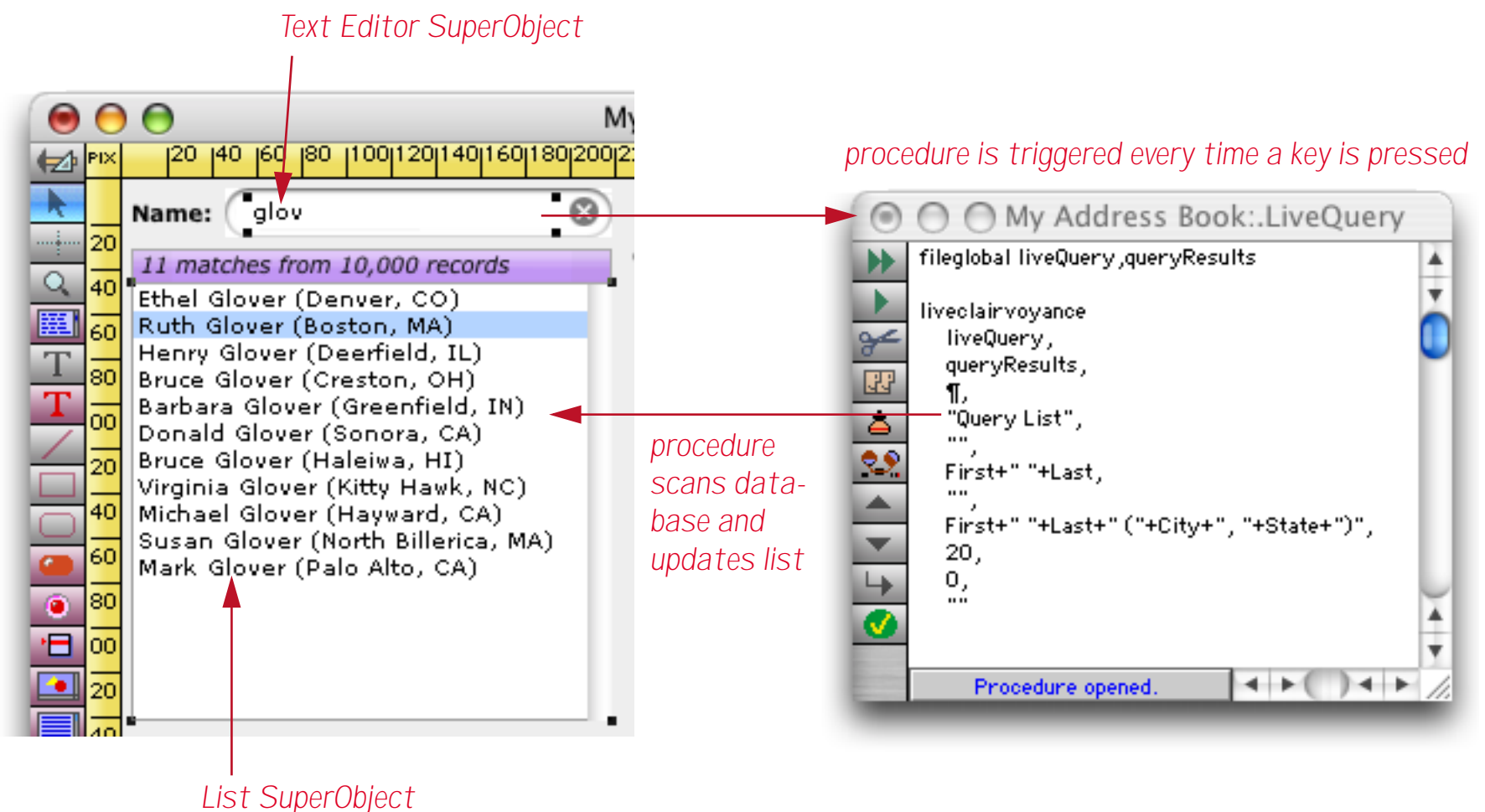


First	Last	Organization	Title	Address	City
Barbara	Moldenhai	Memphis Consultants Inc	CIO	155 Elm Ter	Canandaigua
Beverly	Welsh	Hills Newspapers Limited	Safety Administra	17671 Sandhill Dr	Duluth
Joseph	Diaz	North Agency Assoc.	Senior Applications	891 South Ravenswood Ci	Palm Coast
Peter	Guthrie	Future Insurance Co.	Planning Director	22460 Windsor Pkwy	Boston
Kenneth	Peck	East Semiconductor Institute	Buidlding Maintenai	932 E Fulton St.	Chicago
Francis	Phelps	Northwest Capital Productions	Senior GIS Technic	333 Brookwood Trail	Wilmington
Denise	Orshak	Middle Information Assoc	Senior GIS Technic	8338 East Marshall Apt	Fargo
Anna	Smythe	Mark Newspapers Foundation	Maintenance Mecha	267 West Westminster Ci	Orlando
Helather	Booth	Hamilton Agency Group	Materials Supervis	22477 North Burlington P	Miami
Marvin	Steele	Family Environmental Group	Department Budget	12519 E Ramon Ave	Wellton

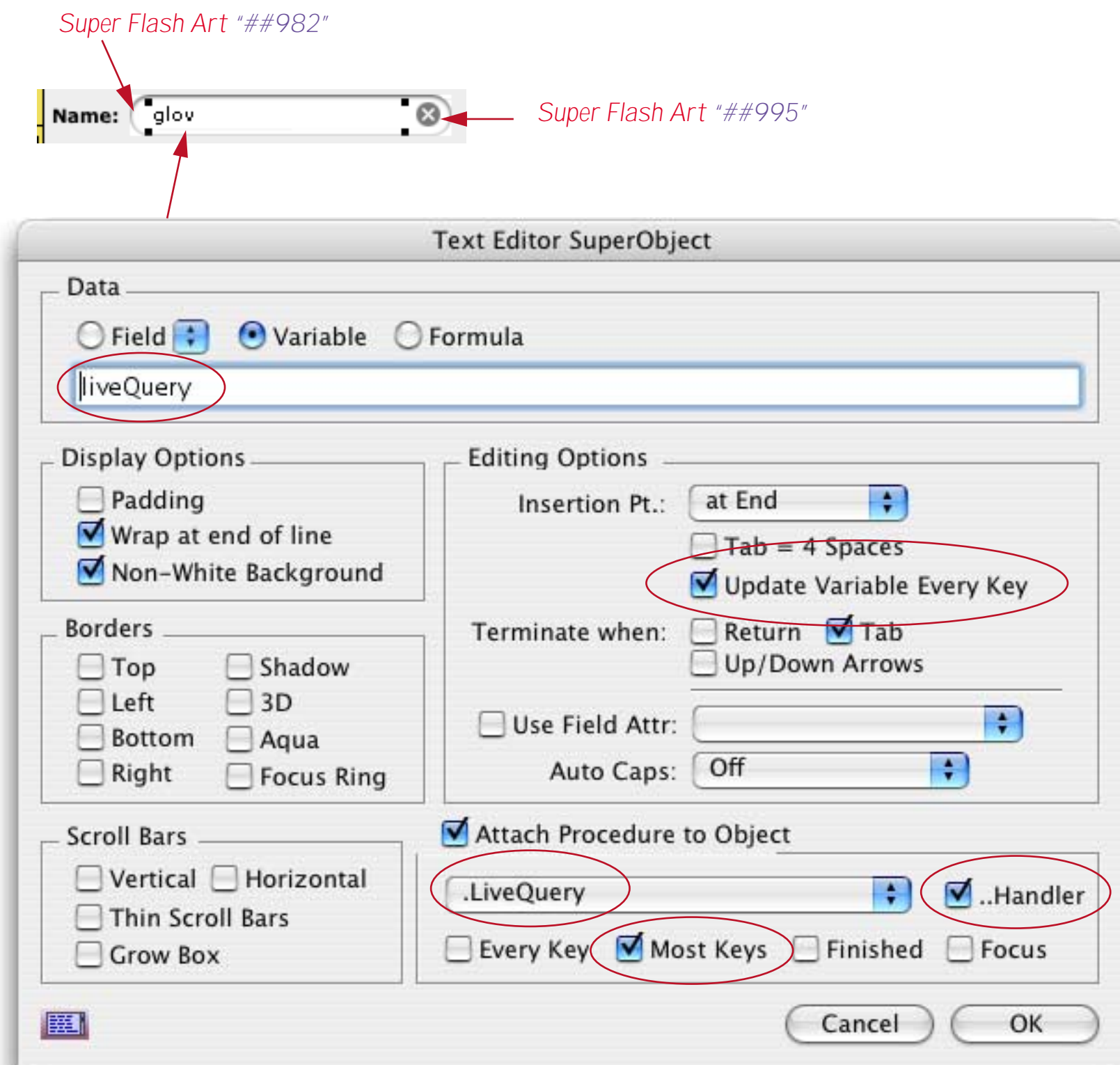
Here is the form that has been set up to use Live Clairvoyance.



This form has many elements, but the Live Clairvoyance portion consists of only three — a Text Editor SuperObject to type in the search word or phrase, a List SuperObject to display the resulting list, and a procedure to perform the actual database scan.



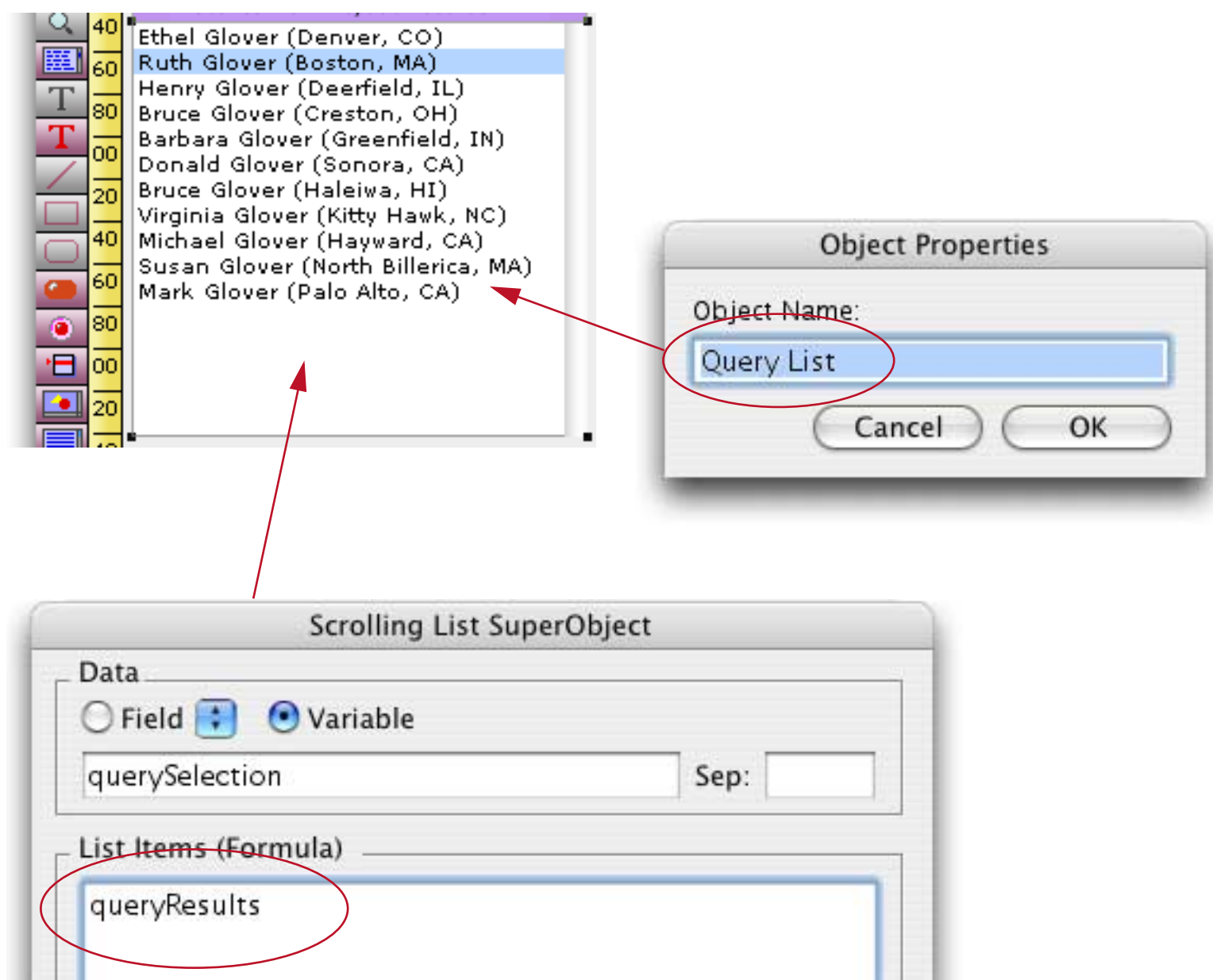
Live Clairvoyance starts with typing in one or more characters to search for. This is done with a Text Editor SuperObject. This illustration shows the options that must be set for this object. (It also shows the formulas for two Super Flash Art objects to display the background and cancel icon for the search area. See the **Dialogs & Icons** wizard for more information on these images.)



The **Data** option is set to a variable named **liveQuery**. You can use any variable name you like, but you'll need to use the same name when you write the procedure that is triggered by this object.

In the **Editing Options** the **Update Variable Every Key** option must be enabled. This allows the procedure to see what you have typed. Moving on to the procedure, in this case it is named **.LiveQuery**. You can use any name you like as long as it is selected in this pop-up menu. You also want to check the **Most Keys** and **..Handler** options.

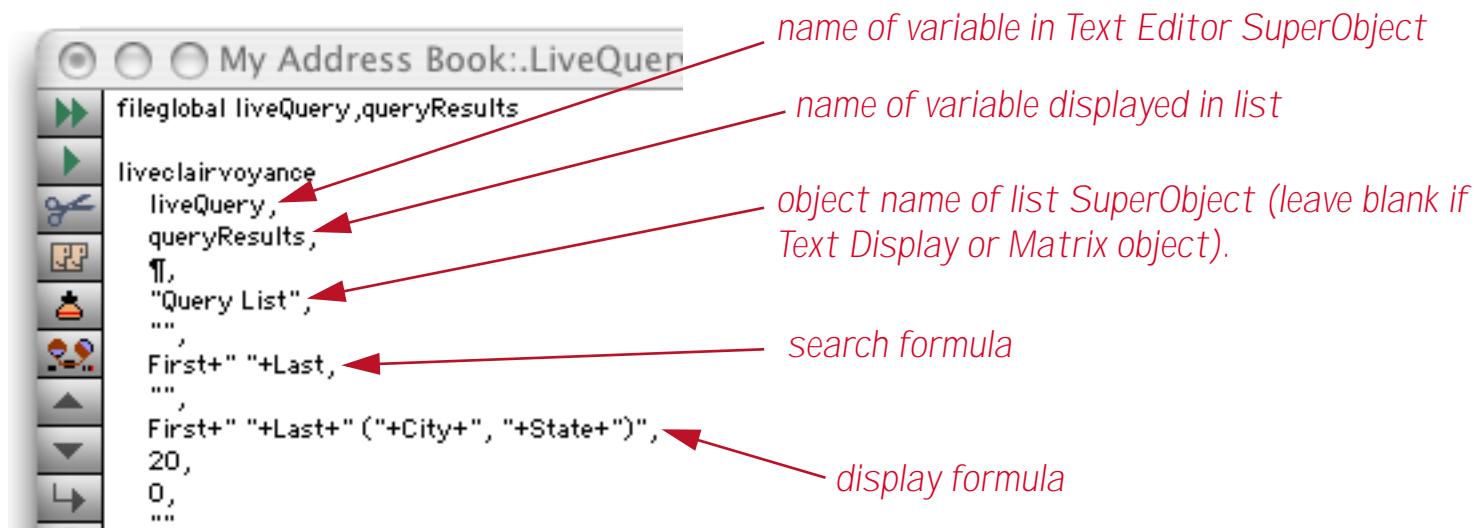
The Live Clairvoyance output can be displayed with a Text Display SuperObject, a Matrix SuperObject, or a List SuperObject, as in this case.



To set up this List object you'll need to set up two dialogs. When first setting up the object you'll need to set the **List Items (Formula)** option to the name of a variable — in this case we picked `queryResults` but again you can use any name you want. (In this example the **Data** option was also set to a variable named `querySelection`, but this variable is not directly involved in the operation of Live Clairvoyance.)

When using a List Object you must also set the name of the object. In this case we set the name to `Query List`, but again you can choose any name you want as long as you remember what the name is so that you can use it in the procedure. It's not necessary to set the object name when displaying the result in a Text Display or Matrix Super Object.

Once the form objects are set up there's only one remaining piece - the procedure.



The procedure only has two statements. The first statement defines the two variables used, **liveQuery** for the search text and **queryResults** for the output list.

The second statement, **liveclairvoyance**, does all the heavy lifting. This statement has eleven parameters.

```
LIVECLAIRVOYANCE INPUT,OUTPUTLIST,SEPARATOR,LISTOBJECT,DATABASE,QUERY,COMPARE,TEMPLATE,
TICKS,MAX,OPTIONS
```

The **Input** and **OutputList** parameter must be set to the variables defined in the Text Editor SuperObject and List SuperObject. The **ListObject** parameter must be set to the name of the list object (if a list object is being used, otherwise set this parameter to "").

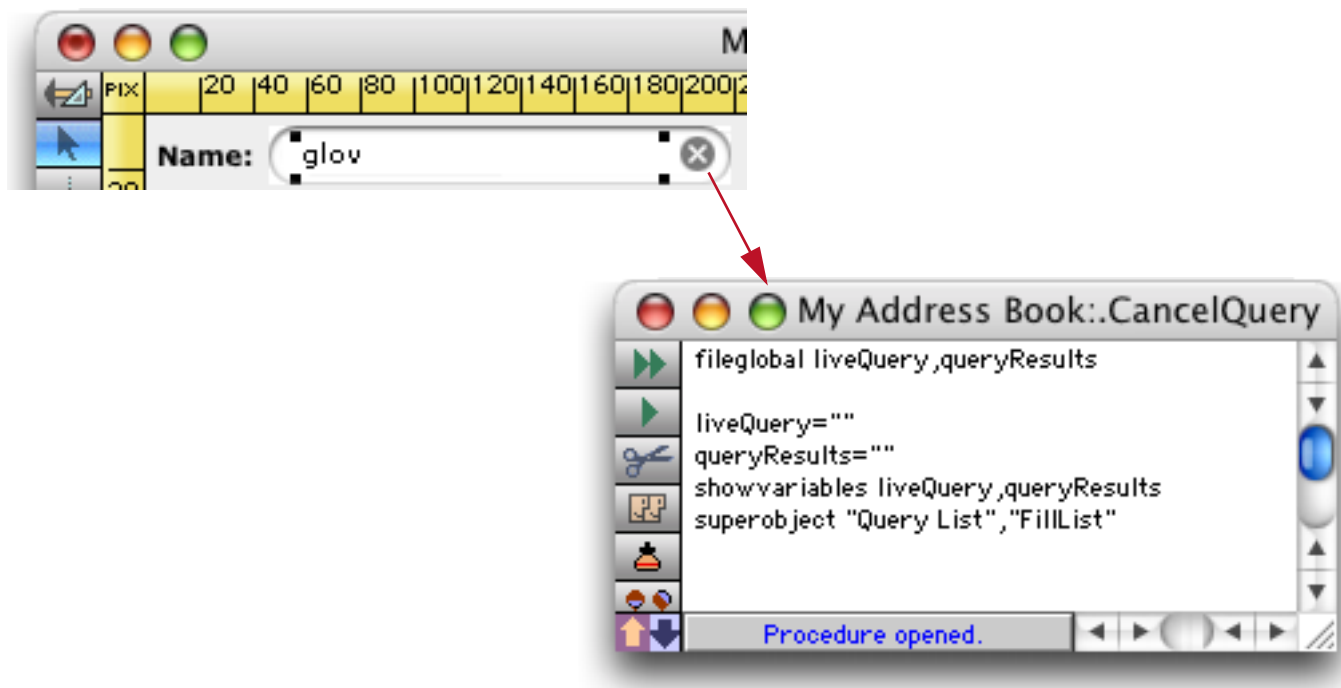
The table below gives detailed explanations for each parameter.

Parameter	Example	Description
Input	liveQuery	This parameter must contain the data to be searched for, usually in a variable.
OutputList	queryResults	This parameter is the name of a variable that will contain the final output array of matching entries.
Separator	¶	The separator character for the output array.
ListObject	"Query List"	If non-blank, this parameter must specify the name of a List SuperObject to be updated each time a search is performed.
Database	""	Name of the database being searched, or blank for the current database.

Parameter	Example	Description
Query	First+" "+Last	Formula used for searching. Each time a search is triggered (usually by pressing a key) the statement will scan the database and check to see if the search text matches this formula. For example, the example formula to the left would find text located in the first and last name fields. This formula must include all of the fields in the database that you want to search (keep in mind, however, that searching more fields may affect performance in large databases).
Compare	""	This parameter specifies the comparison operator to use. The choices are contains, beginswith, endswith, soundslike, match and = (you must put quotes around each of these. If you leave this empty it will default to contains, so in the example database the search performed will be <code>First+" "+Last contains liveQuery</code>
Template	First+" "+Last+" (" +City+", "+State+"")"	This parameter specifies how the fields in the database will be displayed in the output list. In this example the output will contain the first and last names, followed by the city and state within parentheses.
Ticks	20	This parameter specifies how often the display will be updated during the search. The value is specified in ticks (1 tick is equal to 1/60th of a second). A value of 20 will cause the display to update 3 times per second. Up to a point, lower values will make the search "look" faster because partial results appear sooner. If this value is set to zero then no partial results will be displayed, and nothing will appear until the entire search is complete.
Max	0	This is the maximum number of elements that can appear in the output array. Use 0 for no limit.
Options	" "	This parameter can contain one or more options to modify the search. If it contains "sort" the output array will be sorted alphabetically. If it contains "selected" only selected records will be searched, otherwise all records will be searched (even if some are currently not selected).

Adding a Cancel Search Button

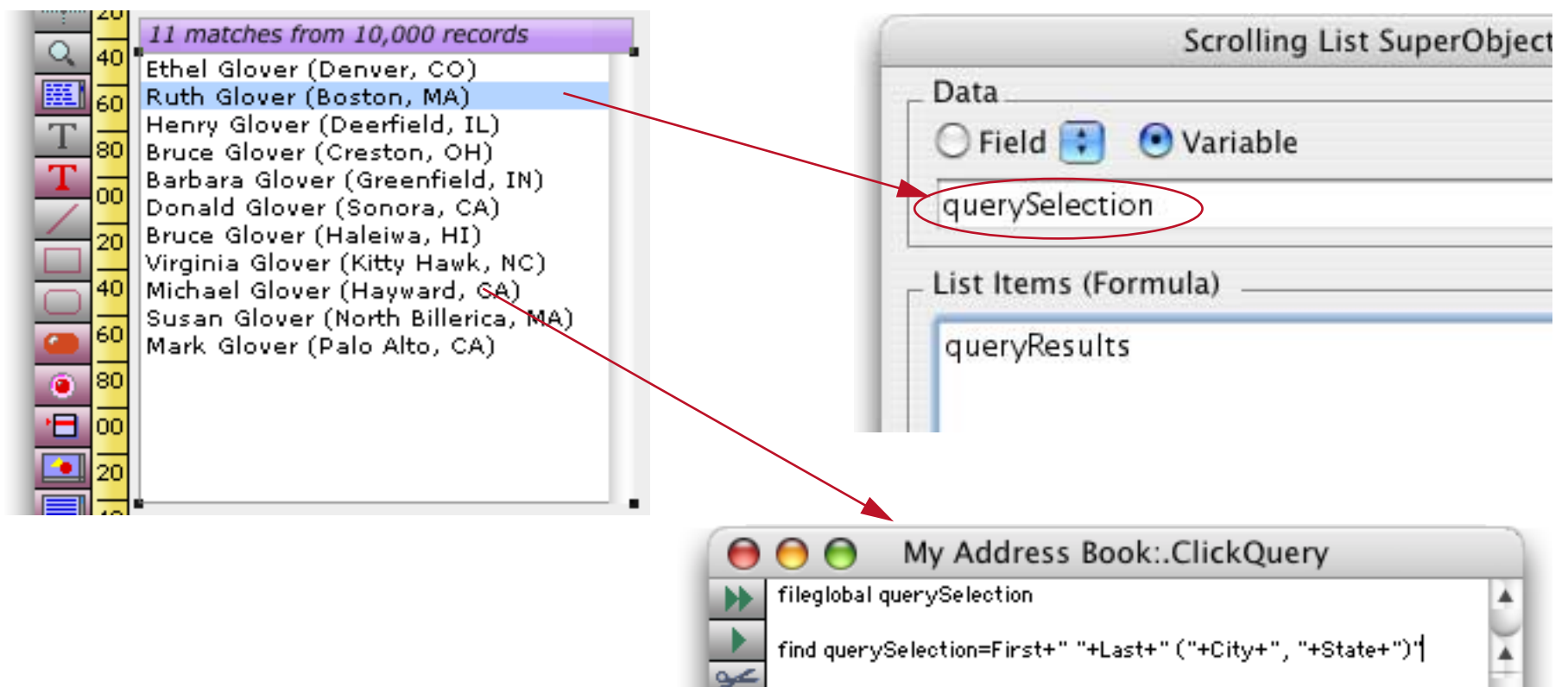
You may want to add a cancel search button. This is easy to do, as shown below.



The procedure simply clears the variables for editing the search text (in this example `liveQuery`) and displaying the search results (in this example `queryResults`), and then displays these variables with the `showvariables` statement. The final line is only required if you are displaying the results with a List Super-Object — it forces the list to display the new (blank) information.

Clicking on the Live Clairvoyance List Object

This form has been set up so that when you click on a line in the list the database jumps to the corresponding record. This allows you to display and/or edit the data. When you click on a line in the Scrolling List Super-Object it will place the value of the line you click on into a variable named `querySelection` and then trigger a procedure named `.ClickQuery` (the pop-up menu for setting this up is not shown).



The procedure simply uses a `find` statement to locate the corresponding record. If you flip back a couple of pages, you'll see that the right side of the equals sign (`First+" "+Last+" (" +City+", "+State+")'`) is the same as the Template formula parameter of the `LiveClairvoyance` statement. As long as these two formulas match this procedure will be able to find the record that matches the item you clicked on.

Summaries and Outlines

Summarizing a database is a three step process — group, calculate and outline (see “[3-Step Summarizing](#)” on page 365 of the *Panorama Handbook*). The table below lists the statements that can be used to automate this process. Each statement corresponds to a menu command or tool. In fact, the easiest way to write a procedure to summarize the database is to simply record it (see “[Creating a Procedure with the Recorder](#)” on page 212 and “[Adding a Recording to an Existing Procedure](#)” on page 223).

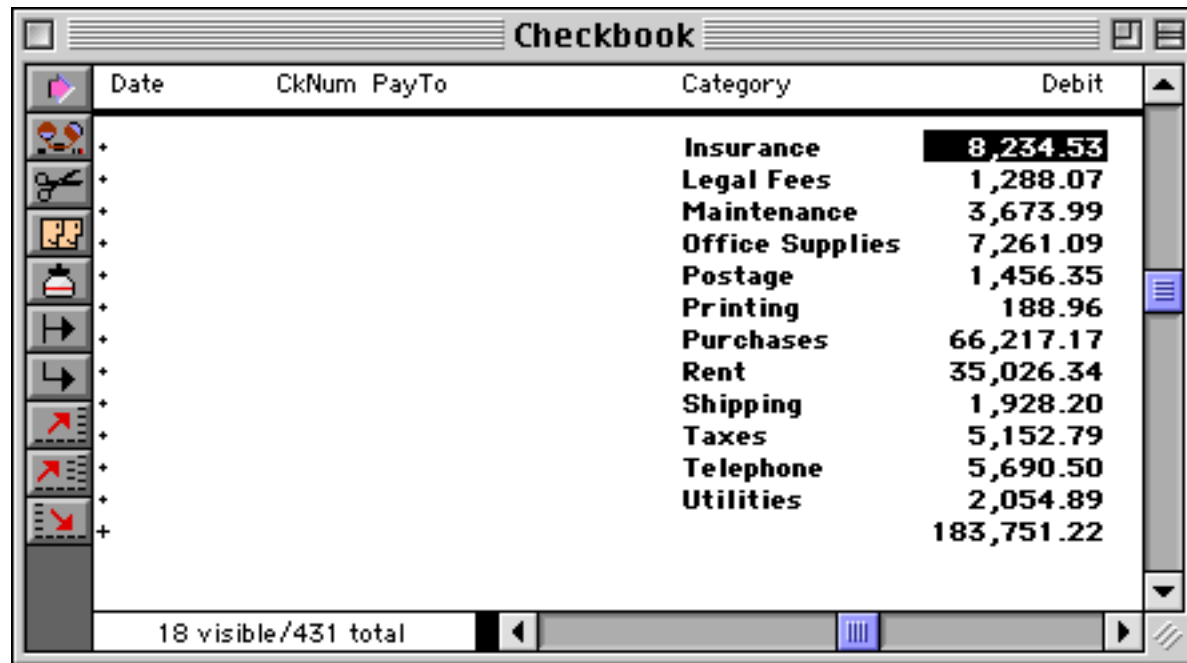
Step	Statement	Ref	Description
Group	groupup	Page 5337	This statement groups the database by the current field (see “ Moving Left and Right ” on page 515). The database is sorted in ascending order (A's to Z's) and a summary recorded is added at each place where the value in the field changes. If the current field is a date field you must add by day, by week, by month, by quarter or by year after the groupup statement.
	groupdown	Page 5336	This statement works exactly like groupup , but sorts the database in descending order (Z's to A's).
	group	Page 5334	This statement groups the database without sorting it.
	groupbycolor	Page 5335	This statement groups the database by color.
Calculate	total	Page 5865	This statement calculates totals and subtotals in the current field (see “ Moving Left and Right ” on page 515).
	average	Page 5069	This statement calculates averages and subaverages in the current field.
	count	Page 5131	This statement counts non-empty values in the current field.
	minimum	Page 5531	This statement calculates minimum values in the current field.
	maximum	Page 5524	This statement calculates maximum values in the current field.
Outline	outlinelevel	Page 5586	This statement expands or collapses the database to show a specific level of detail. The statement has one parameter, which may be "Data" to show all of the detail, or a summary level from "1" to "7" to show a specific outline level.
	removesummaries	Page 5657	This statement removes some or all of the summary records in the database. The statement has one parameter, which specifies the level of summaries to be removed (from "1" to "7"). To make sure that all summary records are removed use "7".
	removedetail	Page 5656	This statement removes the data records from the database, leaving only the summary records. It can also remove lower level summary records. The remaining summary records are dropped in level (the lowest remaining summary records become data records). The statement has one parameter, the lowest level of summary record to be retained (from "1" to "7"). To remove just the data records and leave all summary records use "1".
	collapse	Page 5115	This statement hides any detail records associated with the current summary record.
	expand	Page 5203	This statement expands the next level of detail associated with the current summary record.
	expandall	Page 5205	This statement expands all of the detail associated with the current summary record, right down to the data records.
	info("summary")	Page 5428	This function returns the summary record of the current record, from 0 (data record) to 7 (highest level summary).
	info("expandable")	Page 5374	This function checks to see if the current record is an expandable summary record. It returns false if this is a data record or if this record is already expanded.

Summary/Outline Examples

To summarize a database using a procedure you simply pick one or more statements from each of the three steps — group, calculate and outline. This very basic procedure summarizes a checkbook by category and displays just the totals for each category.

```
field Category      /* STEP 1 - GROUP */
groupup
field Debit         /* STEP 2 - CALCULATE */
total
outlinelevel "1"   /* STEP 3 - OUTLINE */
```

The end result of running this procedure will look something like this.



Date	CkNum	PayTo	Category	Debit
			Insurance	8,234.53
			Legal Fees	1,288.07
			Maintenance	3,673.99
			Office Supplies	7,261.09
			Postage	1,456.35
			Printing	188.96
			Purchases	66,217.17
			Rent	35,026.34
			Shipping	1,928.20
			Taxes	5,152.79
			Telephone	5,690.50
			Utilities	2,054.89
				183,751.22

18 visible/431 total

To remove the summaries and get back to the original data we can use a simple one line procedure.

```
removesummaries "7"
```

This slightly more complex procedure will group the database by month and by category within each month.

```

field Date          /* STEP 1 - GROUP */
groupup by month
field Category
groupup
field Debit         /* STEP 2 - CALCULATE */
total
outlinelevel "1"   /* STEP 3 - OUTLINE */

```

Here is the result of running this procedure.

Date	CkNum	PayTo	Category	Debit
			Advertising	2,841.02
			Equipment Rent:	96.05
			Insurance	761.63
			Legal Fees	223.52
			Office Supplies	426.93
			Postage	150.00
			Purchases	17,083.42
			Rent	4,070.83
			Shipping	231.72
			Taxes	549.00
			Telephone	141.09
+01/31/99				26,575.21
				0.00
			Advertising	8,134.13
			Auto	240.21
			Equipment Rent:	73.14
			Fixed Assets	428.39
			Insurance	601.48
			Legal Fees	95.00
			Maintenance	310.00
			Office Supplies	915.50
			Postage	315.00
			Purchases	3,386.78
			Rent	7,742.19
			Shipping	192.00
			Telephone	736.66
			Utilities	402.78
+02/28/99				23,573.26

If the last line of the procedure had been

```
outlinelevel "2"
```

Then the final result would have looked like this.

Date	CkNum	PayTo	Category	Debit
+01/31/99				26,575.21
+02/28/99				23,573.26
+03/30/99				39,011.30
+04/27/99				5,852.73
+05/31/99				27,659.93
+06/28/99				12,742.03
+07/25/99				18,860.65
+08/29/99				14,842.86
+09/28/99				14,533.25
+08/02/00				100.00
+				183,751.22

As the procedure runs it flashes the window over and over again. To eliminate this you can use the `noshow` and `endnoshow` statements (see “[Suppressing Display of Text and Graphics](#)” on page 307).

```

noshow
  field Date          /* STEP 1 - GROUP */
  groupup by month
  field Category
  groupup
  field Debit        /* STEP 2 - CALCULATE */
  total
  outlinelevel "1"  /* STEP 3 - OUTLINE */
  showpage
endnoshow

```

This revised procedure will only re-display the window once, at the very end.

Calculating Grand Totals

There are two methods for calculating a grand total without subtotals. The first is to simply use the `total` statement. This adds a single summary record at the bottom of the database and calculates the total. (You can also use the `average`, `count`, `minimum` and `maximum` statements this way.)

```

field Debit
total

```

This procedure produces a single summary record with the total, like this.

The screenshot shows a window titled 'Checkbook' containing a table of transactions. The table has columns for Date, CkNum, PayTo, Category, and Debit. The last row is a summary row with a total of 183,751.22. The status bar at the bottom indicates '413 visible/413 total'.

Date	CkNum	PayTo	Category	Debit
09/10/99	2268	Cable & Wireless	Telephone	120.16
09/18/99	2276	PacTel Cellular	Telephone	398.19
09/19/99	2287	Cable & Wireless	Telephone	124.03
09/19/99	2289	G T E	Telephone	349.50
09/19/99	2288	MCI	Telephone	67.59
09/19/99	2290	City Of Caboose	Utilities	103.15
09/19/99	2291	S C E	Utilities	81.13
09/19/99	2292	So. Calif. Gas Co.	Utilities	154.95
08/02/00	2307			100.00
				183,751.22

Another method for calculating a grand total is to use the `formulasum` statement. This statement has two parameters:

```

formulasum result,formula

```

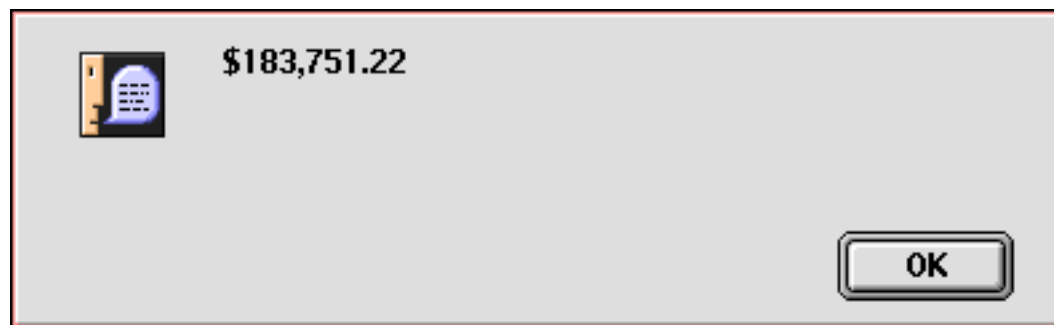
The `result` parameter must be a field or variable. The final total will be stored in here.

The `formula` parameter is a formula that will be evaluated for every selected record. Starting from the top of the database, Panorama will visit each record and calculate the result of the formula. As it goes it keeps a running total of the results. The final result is the sum of all of the individual results for each selected record.

The procedure below calculates the grand total for the checkbook database and displays it. The database itself is not modified (no summary record is added).

```
local total
formulasum total,Debit
message pattern(total,"$#,##")
```

The procedure will display the total like this.



By changing the formula you can calculate different sums. This procedure calculates the number of checks that are over \$500.

```
local count
formulasum count,?(Debit>500,1,0)
message "There are "+str(count)+" checks over $500."
```

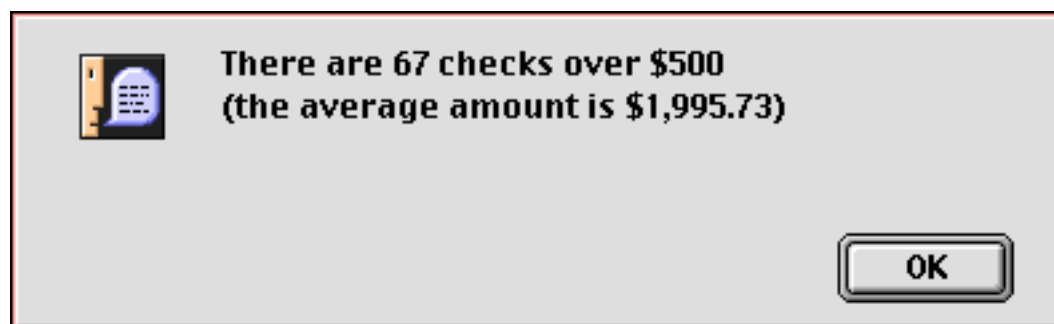
Here is the result.



There's no reason you can't use **formulasum** more than once, like this.

```
local count,total
formulasum count,?(Debit>500,1,0)
formulasum total,?(Debit>500,Debit,0)
message "There are "+str(count)+
  " checks over $500"+¶+"(the average amount is "+
  pattern(total/count,"$#.,##")+")"
```

This procedure displays both the number of checks over \$500 and the average value of these checks.



Running Total

The `runningtotal` statement performs a special computation. Unlike the other summary calculations, this statement modifies every data cell in the currently active field, not just the summary records. Like the `total` statement, `runningtotal` starts at the top of the database and adds up each data cell as it moves down the column. The `runningtotal` statement, however, replaces each data cell with the current total. The result is a field which contains the cumulative total at each point in the database. Here is a procedure that uses `runningtotal` to calculate a checkbook's balance after each transaction.

```
noshow
  field Balance
  formulafill Credit-Debit
  runningtotal
  showcolumns Balance
endnoshow
```

To see what the result of this procedure looks like go to "[Using Running Total to Balance a Checkbook](#)" on page 399 of the *Panorama Handbook*. By the way, this procedure uses the `noshow`, `showcolumns` and `endnoshow` statements to make sure that the window only get's redisplayed once. These statements are not necessary for the procedure to operate, but do make it look a little bit cleaner as it runs ("[Suppressing Display of Text and Graphics](#)" on page 307).

Running Difference

The `runningdifference` statement is the opposite of `runningtotal`. This statement fills each data cell with the difference between the cell and the cell above it. Use the `runningdifference` statement when you want to calculate the spread or interval between consecutive values, for example odometer readings or dates. Here is a procedure that uses `runningdifference` to calculate gas mileage per gallon for each fill-up.

```
field Range
formulafill Odometer
runningdifference
field MPG
formulafill Range/Gallons
```

Go to "[Using Running Difference to Calculate Gas Mileage](#)" on page 403 of the *Panorama Handbook* to see what the result of this procedure looks like.

Transforming Big Chunks of Data

The **Math** menu contains ten commands for transforming an entire field of data at once (see “[Data Processing](#)” on page 433 of the *Panorama Handbook*). All of these statements operated on the current field, so you’ll need to position the current field before you use them (see “[Moving Left and Right](#)” on page 515). In addition these commands only operate on selected data, so you must make sure that the proper data is selected before the statement is used (see “[Selecting Information](#)” on page 557).

The workhorse of this group is **formulafill**, which you will probably use more than all the others combined.

Category	Statement	Ref	Description
Fills	formulafill	Page 5274	This statement fills all of the selected cells in the current field with a formula. Panorama starts at the top of the database and works its way down, calculating a the formula result over and over again for each record. See “ Starting with a Formula ” on page 439 of the <i>Panorama Handbook</i> .
	fill	Page 5243	This statement also fills all of the selected cells in the current field with a formula. Unlike formulafill , however, this statement calculates the formula only once, before it starts scanning the database. If you are filling the field with a constant value like “US Mail” this statement will be slightly faster than the formulafill statement.
	emptyfill	Page 5192	This statement fills all empty cells in a field with a formula. Like the fill statement, the formula is only calculated once before Panorama begins scanning the database.
	sequence	Page 5728	This statement fills a numeric field with an increasing or decreasing numeric sequence, for example 1, 2, 3 or 100, 99, 98.
	change	Page 5095	This statement scans the current field searching for a word or phrase. It replaces every occurrence of the word or phrase it finds with another word or phrase. See “ Change (Find and Replace) ” on page 472 of the <i>Panorama Handbook</i> .
Propagates	propagate	Page 5619	This statement copies the values in the current field into the empty cells (if any) below. See “ Propagate ” on page 466 of the <i>Panorama Handbook</i> .
	unpropagate	Page 5874	This statement is the opposite of propagate . It scans the database from top to bottom. If it finds the same value two or more times in a row it erases all but the topmost duplicate value. See “ UnPropagate ” on page 469 of the <i>Panorama Handbook</i> .
	propagateup	Page 5620	This statement copies the values in the current field into the empty cells (if any) above. See “ Propagate ” on page 466 of the <i>Panorama Handbook</i> .
	unpropagateup	Page 5875	This statement is the opposite of unpropagate . It scans the database from bottom to top. If it finds the same value two or more times in a row it erases all but the bottommost duplicate value. See “ UnPropagate ” on page 469 of the <i>Panorama Handbook</i> .
Appearance	stylecolor	Page 5810	This statement changes the style (bold, italic, etc.) and color (red, green, blue, etc.) of one or more data cells. See “ Data Style and Color ” on page 474 of the <i>Panorama Handbook</i> .

Making Transformations Even Faster

Panorama's transformation (`formulafill`, `propagate`, etc.) statements operate very quickly, even when used with large databases. However, the `noundo` statement can make these operations even faster. This statement disables Panorama's undo feature. Since the transformation doesn't need to worry about undo it can run slightly faster. Here is an example of how to use `noundo` in a procedure with the `formulafill` statement.

```
noundo
field Total
formulafill A+B+C+D
field Avg
formulafill (A+B+C+D)/4
```

The benefit of the `noundo` statement will not be noticeable on smaller databases, but becomes more pronounced the larger the database gets.

Numeric Calculations with FormulaFill

On numeric fields the `formulafill` statement can be used to calculate totals, averages, discounts, percentages, etc. The statement must be followed by a formula to calculate (see "[Formulas](#)" on page 19). To illustrate this statement we'll use this database.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93		
Bakersfield	2.29	8.26	12.91	3.02		
Camarillo	2.83	9.19	15.11	2.54		
Diamond Bar	3.13	7.81	13.19	3.11		
Fullerton	2.59	8.25	13.48	1.77		
Laguna Beach	3.06	9.45	12.5	3.01		
Newport Beach	3.18	7.22	12.32	2.38		
Whittier	3.67	5.23	14.24	3.27		

This procedure will calculate all the values in the total and average fields.

```
field Total
formulafill A+B+C+D
field Avg
formulafill (A+B+C+D)/4
```

Here's the finished result.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13	7.81	13.19	3.11	27.24	6.81
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.10	6.27
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

Suppressing Zero's

If a new record with incomplete information is added to the database the empty values are treated as zeroes, as shown here.

City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13	7.81	13.19	3.11	27.24	6.81
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.10	6.27
Santa Ana					0.00	0.00
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

Sometimes you may want a zero result to be suppressed, leaving the cell blank. To do this use the **zeroblank()** function, like this (see "[Suppressing Zero's](#)" on page 574).

```
field Total
formulafill zeroblank(A+B+C+D)
field Avg
formulafill zeroblank((A+B+C+D)/4)
```

When this procedure is run any zero values are treated as blanks.

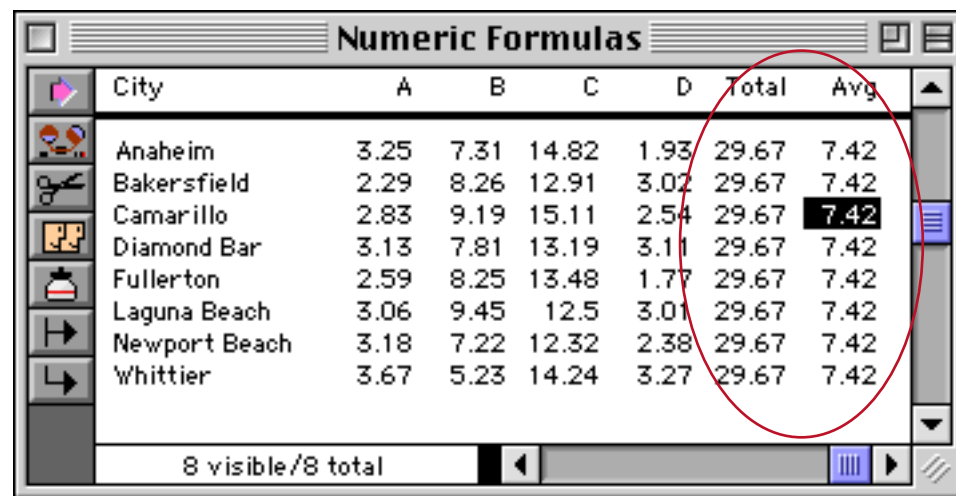
City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	27.31	6.83
Bakersfield	2.29	8.26	12.91	3.02	26.48	6.62
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13	7.81	13.19	3.11	27.24	6.81
Fullerton	2.59	8.25	13.48	1.77	26.09	6.52
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	25.10	6.27
Santa Ana						
Whittier	3.67	5.23	14.24	3.27	26.41	6.60

Fill vs. FormulaFill

Like the `formulafill` statement, the `fill` statement also takes a formula and fills all the cells in the current column with the result. However, there is a big difference. The `formulafill` statement calculates the formula over and over again, producing a separate result for every record. The `fill` statement only calculates the formula once, before it starts. It then fills all the cells with the same value. To illustrate we'll use a modified version of the procedure from the last section.

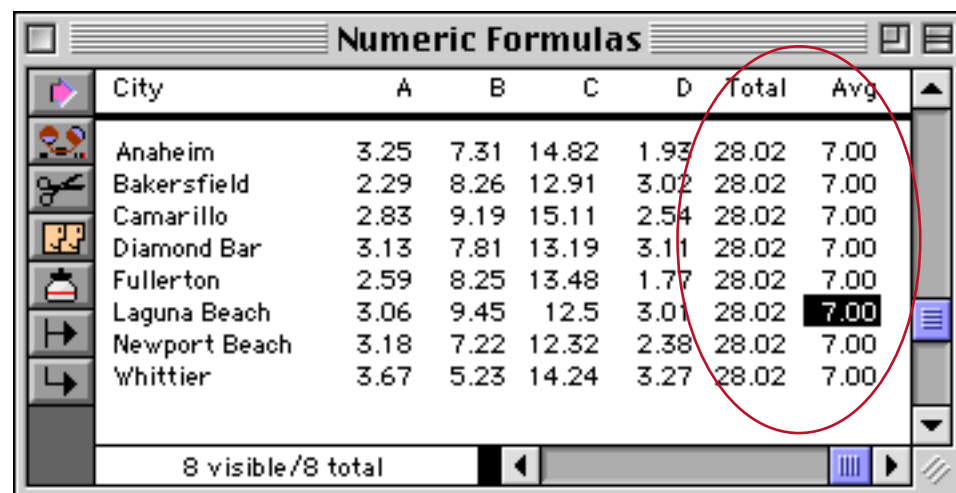
```
field Total
fill zeroblank(A+B+C+D)
field Avg
fill zeroblank((A+B+C+D)/4)
```

The result of this procedure depends on what record is active. In this case every cell is filled with the total and average for **Camarillo**.



City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	29.67	7.42
Bakersfield	2.29	8.26	12.91	3.02	29.67	7.42
Camarillo	2.83	9.19	15.11	2.54	29.67	7.42
Diamond Bar	3.13	7.81	13.19	3.11	29.67	7.42
Fullerton	2.59	8.25	13.48	1.77	29.67	7.42
Laguna Beach	3.06	9.45	12.5	3.01	29.67	7.42
Newport Beach	3.18	7.22	12.32	2.38	29.67	7.42
Whittier	3.67	5.23	14.24	3.27	29.67	7.42

If we click on **Laguna Beach** and run the procedure again we'll get a different result.

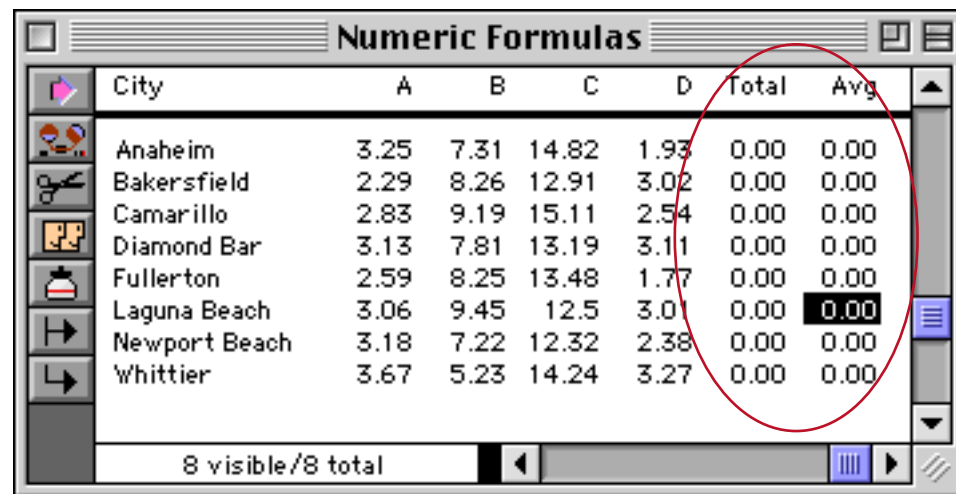


City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	28.02	7.00
Bakersfield	2.29	8.26	12.91	3.02	28.02	7.00
Camarillo	2.83	9.19	15.11	2.54	28.02	7.00
Diamond Bar	3.13	7.81	13.19	3.11	28.02	7.00
Fullerton	2.59	8.25	13.48	1.77	28.02	7.00
Laguna Beach	3.06	9.45	12.5	3.01	28.02	7.00
Newport Beach	3.18	7.22	12.32	2.38	28.02	7.00
Whittier	3.67	5.23	14.24	3.27	28.02	7.00

The `fill` statement works fine for constant values, and is slightly faster than `formulafill`.

```
field Total
fill 0
field Avg
fill 0
```

This procedure fills the columns with zeroes.

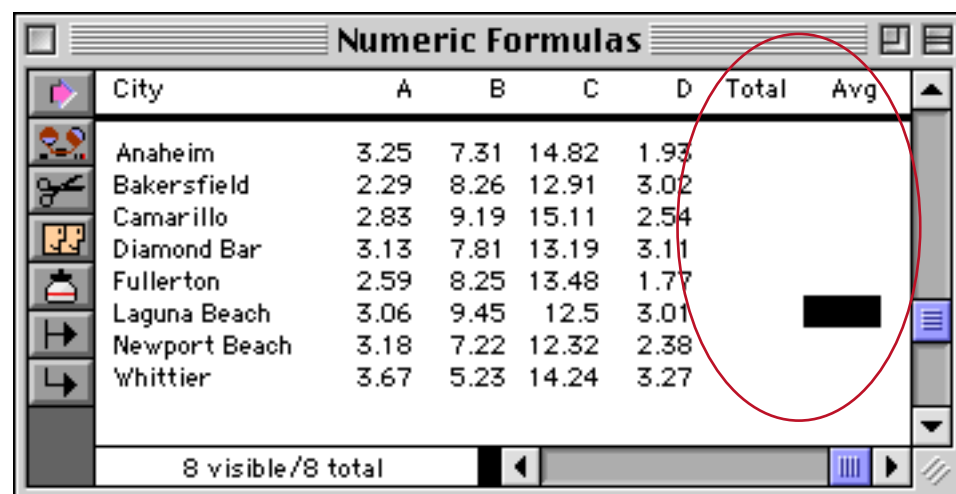


City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93	0.00	0.00
Bakersfield	2.29	8.26	12.91	3.02	0.00	0.00
Camarillo	2.83	9.19	15.11	2.54	0.00	0.00
Diamond Bar	3.13	7.81	13.19	3.11	0.00	0.00
Fullerton	2.59	8.25	13.48	1.77	0.00	0.00
Laguna Beach	3.06	9.45	12.5	3.01	0.00	0.00
Newport Beach	3.18	7.22	12.32	2.38	0.00	0.00
Whittier	3.67	5.23	14.24	3.27	0.00	0.00

Use the `zeroblank()` function if you want to fill a numeric field with zeroes (see “[Suppressing Zero's](#)” on page 574).

```
field Total
fill zeroblank(0)
field Avg
fill zeroblank(0)
```

The result is two completely empty columns.



City	A	B	C	D	Total	Avg
Anaheim	3.25	7.31	14.82	1.93		
Bakersfield	2.29	8.26	12.91	3.02		
Camarillo	2.83	9.19	15.11	2.54		
Diamond Bar	3.13	7.81	13.19	3.11		
Fullerton	2.59	8.25	13.48	1.77		
Laguna Beach	3.06	9.45	12.5	3.01		
Newport Beach	3.18	7.22	12.32	2.38		
Whittier	3.67	5.23	14.24	3.27		

By the way, you could get the same result with the `formulafill` statement. It would be slightly slower, but the difference would not be measurable unless the database contained tens of thousands of records.

Using FormulaFill to Transform Text

Using the `formulafill` statement you can combine multiple fields, split a field apart, re-arrange words or phrases, and translate characters (for example, converting uppercase to lower case). To illustrate a few examples of this we'll use this contacts database.

First	Last	Name	Title	Company	Address
John	Smith		Sales Manager	Acme Widgets	12 Harmony Lane
James	Mahan		Owner	J.M. Plumbing	1294 W. 31st
Thom	Getchell		Customer Support	Thom's Appliances	543 Laurel
Lee	Tucker		Sales Manager	Latham Video	4792 Latham
Jim	Nickle		President	Jim's Appliances	14189 8th
Susan	Brown				783 Algonquin
Logan	Nourse		Purchasing	Palo Alto Lumber	1828 Amaranta
Mary	Doyle				519 Leahy
Bud	Roble			Riverside Lumber	1901 Red Oak Dr
Jim	Pyle				7265 Lakeland Dr

This example combines the first and last names into a single field.

```
field Name
formulafill First+" "+Last
```

When triggered the result of this procedure looks like this.

First	Last	Name	Title	Company	Address
John	Smith	John Smith	Sales Manager	Acme Widgets	12 Harmony Lane
James	Mahan	James Mahan	Owner	J.M. Plumbing	1294 W. 31st
Thom	Getchell	Thom Getchell	Customer Support	Thom's Appliances	543 Laurel
Lee	Tucker	Lee Tucker	Sales Manager	Latham Video	4792 Latham
Jim	Nickle	Jim Nickle	President	Jim's Appliances	14189 8th
Susan	Brown	Susan Brown			783 Algonquin
Logan	Nourse	Logan Nourse	Purchasing	Palo Alto Lumber	1828 Amaranta
Mary	Doyle	Mary Doyle			519 Leahy
Bud	Roble	Bud Roble		Riverside Lumber	1901 Red Oak Dr
Jim	Pyle	Jim Pyle			7265 Lakeland Dr

This example combines the first and last names in a different way.

```
field Name
formulafill upper(Last)+", "+First
```

When triggered the result of this procedure looks like this.

First	Last	Name	Title	Company	Address
John	Smith	SMITH, John	Sales Manager	Acme Widgets	12 Harmony Lane
James	Mahan	MAHAN, James	Owner	J.M. Plumbing	1294 W. 31St
Thom	Getchell	GETCHELL, Thom	Customer Supp	Thom's Appliances	543 Laurel
Lee	Tucker	TUCKER, Lee	Sales Manager	Latham Video	4792 Latham
Jim	Nickle	NICKLE, Jim	President	Jim's Appliances	14189 8th
Susan	Brown	BROWN, Susan			783 Algonquin
Logan	Nourse	NOURSE, Logan	Purchasing	Palo Alto Lumber	1828 Amaranta
Mary	Doyle	DOYLE, Mary			519 Leahy
Bud	Roble	ROBLE, Bud		Riverside Lumber	1901 Red Oak Dr
Jim	Pyle	PYLE, Jim			7265 Lakeland D

Use text funnels to split a field apart or to re-arrange words or phrases. Text funnels allow a formula to extract part of a cell based on a fixed character position within the cell, or based on patterns and context within the cell. See “[Taking Strings Apart \(Text Funnels\)](#)” on page 69 for a complete explanation of text funnels. This procedure will fill the Name field with the first initial and the last name.

```
field Name
formulafill First[1,1]+". "+Last
```

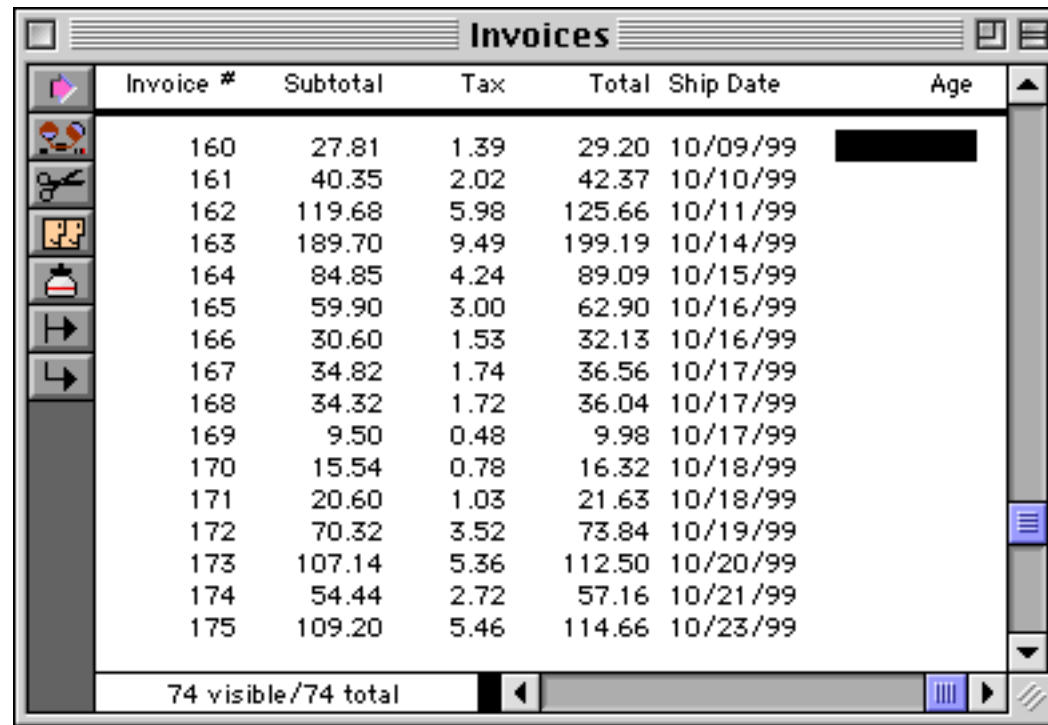
Here is the result.

First	Last	Name	Title	Company	Address
John	Smith	J. Smith	Sales Manager	Acme Widgets	12 Harmony Lane
James	Mahan	J. Mahan	Owner	J.M. Plumbing	1294 W. 31St
Thom	Getchell	T. Getchell	Customer Supp	Thom's Appliances	543 Laurel
Lee	Tucker	L. Tucker	Sales Manager	Latham Video	4792 Latham
Jim	Nickle	J. Nickle	President	Jim's Appliances	14189 8th
Susan	Brown	S. Brown			783 Algonquin
Logan	Nourse	L. Nourse	Purchasing	Palo Alto Lumber	1828 Amaranta
Mary	Doyle	M. Doyle			519 Leahy
Bud	Roble	B. Roble		Riverside Lumber	1901 Red Oak Dr
Jim	Pyle	J. Pyle			7265 Lakeland D

For more information on formulas that take apart and put together text see “[Text Formulas](#)” on page 67.

Date Calculations with Formula Fill

Use the `formulafill` statement to calculate the difference between dates, or to adjust dates. See “[HTML Generating Functions](#)” on page 105 for details on performing calculations with dates. A typical use for date arithmetic is aging of an accounts receivable database.



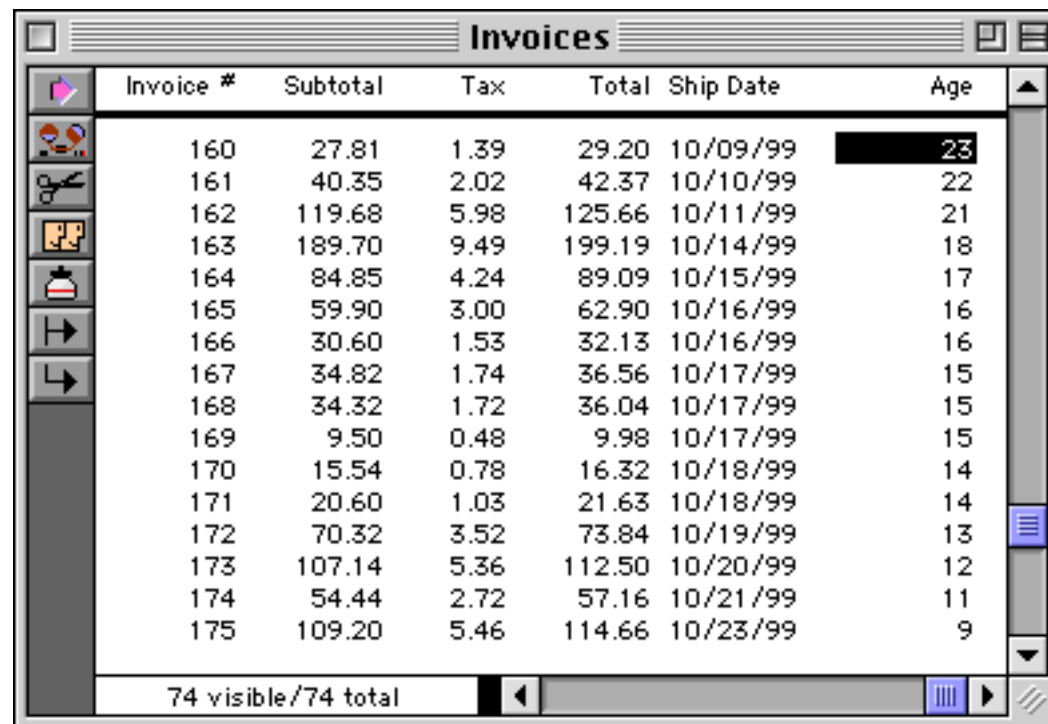
Invoice #	Subtotal	Tax	Total	Ship Date	Age
160	27.81	1.39	29.20	10/09/99	
161	40.35	2.02	42.37	10/10/99	
162	119.68	5.98	125.66	10/11/99	
163	189.70	9.49	199.19	10/14/99	
164	84.85	4.24	89.09	10/15/99	
165	59.90	3.00	62.90	10/16/99	
166	30.60	1.53	32.13	10/16/99	
167	34.82	1.74	36.56	10/17/99	
168	34.32	1.72	36.04	10/17/99	
169	9.50	0.48	9.98	10/17/99	
170	15.54	0.78	16.32	10/18/99	
171	20.60	1.03	21.63	10/18/99	
172	70.32	3.52	73.84	10/19/99	
173	107.14	5.36	112.50	10/20/99	
174	54.44	2.72	57.16	10/21/99	
175	109.20	5.46	114.66	10/23/99	

74 visible/74 total

To calculate the age of an invoice based on the current date, use the **Formula Fill** command with the formula shown here:

```
field Age
formulafill today()-«Ship Date»
```

Here is the result of this procedure.



Invoice #	Subtotal	Tax	Total	Ship Date	Age
160	27.81	1.39	29.20	10/09/99	23
161	40.35	2.02	42.37	10/10/99	22
162	119.68	5.98	125.66	10/11/99	21
163	189.70	9.49	199.19	10/14/99	18
164	84.85	4.24	89.09	10/15/99	17
165	59.90	3.00	62.90	10/16/99	16
166	30.60	1.53	32.13	10/16/99	16
167	34.82	1.74	36.56	10/17/99	15
168	34.32	1.72	36.04	10/17/99	15
169	9.50	0.48	9.98	10/17/99	15
170	15.54	0.78	16.32	10/18/99	14
171	20.60	1.03	21.63	10/18/99	14
172	70.32	3.52	73.84	10/19/99	13
173	107.14	5.36	112.50	10/20/99	12
174	54.44	2.72	57.16	10/21/99	11
175	109.20	5.46	114.66	10/23/99	9

74 visible/74 total

If you want to calculate ages rounded to the nearest 30 day interval use the procedure below instead.

```
field Age
formulafill round((today()-«Ship Date»)-15,30)
```

Here's the result. The age of each invoice rounded to the nearest 30 days.

Invoice #	Subtotal	Tax	Total	Ship Date	Age
101	174.59	8.73	183.32	08/08/99	60
102	130.53	6.53	137.06	08/10/99	60
105	169.65	8.48	178.13	08/14/99	60
106	177.70	8.89	186.59	08/17/99	60
110	144.72	7.24	151.96	08/20/99	60
113	191.38	9.57	200.95	08/24/99	60
115	293.63	14.68	308.31	08/29/99	60
119	122.89	6.14	129.03	09/01/99	60
122	115.65	5.78	121.43	09/04/99	30
123	125.45	6.27	131.72	09/06/99	30
124	155.08	7.75	162.83	09/08/99	30
126	368.73	18.44	387.17	09/14/99	30
130	126.20	6.31	132.51	09/17/99	30
141	104.41	5.22	109.63	09/23/99	30
144	131.70	6.59	138.29	09/25/99	30
146	152.10	7.61	159.71	09/28/99	30
153	141.15	7.06	148.21	10/05/99	0
162	119.68	5.98	125.66	10/11/99	0
163	189.70	9.49	199.19	10/14/99	0
173	107.14	5.36	112.50	10/20/99	0
175	109.20	5.46	114.66	10/23/99	0

21 visible/74 total

For more information on the `today()` and `round()` functions see "[TODAY\(\)](#)" on page 5862 and "[ROUND\(\)](#)" on page 5682 of the *Panorama Reference*.

The SEQ Function

The `seq()` function is a special function for use with the `formulafill` statement. This function returns a unique number for each selected record, starting with 1 at the top of the database. Use this function if you need a unique record number in a formula. Here is an example that fills a column with the words **One**, **Two**, **Three**, **Four**, etc.

```
field Place
formulafill pattern(seq(),"$")
```

When you press **OK** the field is filled in (see "[Displaying Numbers as Words](#)" on page 254 for more information on this output pattern.)

Name	Time	Place	Medal	Behind
Steven Martin	18.17	One		
Joshua Trask	18.19	Two		
Jim Lederman	18.63	Three		
Michael Williams	19.21	Four		
John Wilcox	19.23	Five		
Stan Damone	20.05	Six		
Keith Dawson	20.34	Seven		
Ben Longworth	20.93	Eight		

8 visible/8 total

Here is another example that uses the `seq()` function to assign medals to the first three finishers in the race.

```
field Time
sortup
field Place
formulafill array("Gold/Silver/Bronze",seq(),"/")
```

The first three finishers are assigned gold, silver, and bronze medals, with all of the other records left blank.

Name	Time	Place	Medal	Behind
Steven Martin	18.17	One	Gold	
Joshua Trask	18.19	Two	Silver	
Jim Lederman	18.63	Three	Bronze	
Michael Williams	19.21	Four		
John Wilcox	19.23	Five		
Stan Damone	20.05	Six		
Keith Dawson	20.34	Seven		
Ben Longworth	20.93	Eight		

See “[Text Arrays](#)” on page 93 for more information on the `array()` function used in this example.

Filling Empty Cells

The `emptyfill` statement is very similar to the `fill` statement (see “[Fill vs. FormulaFill](#)” on page 575). However, the `emptyfill` statement will not destroy the data already in the field. In fact, `emptyfill` will only fill cells that are completely empty. Here is a database where some of the name prefixes have been left blank.

T	First Name	Last Name	Company	Street Address	Suite Box	City	State
	Jim	Abrahms	International Transportat	329 North State		Alameda	CA
Ms.	Isabel	Alston	Leader Systems	10463 N. Blaney		San Rafael	CA
	Anthony	Campbell	Drake Inc.	3452 Van Ness		Chicago	IL
Dr.	Robert	Churchill	Smiley & Sons Developme	445 Hanilton		Manasquan	NJ
Ms.	Bea	Clairmont	Arrow Dev.	411 Pacific		Cambridge	MA
Mrs.	Dottie	Davidson	Van Ness Marketing	181 Taintor Dr.		Boston	MA
Mrs.	Barbara	Elliott	Foltene	10440 Torre A		Cambridge	MA
	Lew	Farrell	Coast Label & Supply	33095 Lexingto		San Rafael	CA
Ms.	Kris	Frazee	Mariani Publishing	18550 Lark Ave		San Francisco	CA
	Tim	Hill	Alameda Escrow	1000 Roche Blv		Santa Ana	CA
	Keith	Jacobs	Bath Co.	2994 Garcia Av		Burbank	CA
	Joe	Jones	Professionals Inc.	2 Owen St.		Provo	UT
	Al	Keizer	Certified Labs	1184 Lincoln Av	Suite	Westlake Village	CA
Ms.	Robin	Knight	Fico Appliance Service	2155 S. Bascom	Suite	Tallahassee	FL
	Mark	Lauing	Sherman-Davis	490 Broadway		Naples	FL
Dr.	Bill	Lieber	Arlington Associates	573 Dundee Rd.		Waldwick	NJ
	Russ	Malone	Northridge Bakeries	2210 Wilshire E	Suite	Chicago	IL
Ms.	Jan	Morgan	McCormick-Ridder	4044 Civic Terr		San Diego	CA
	Frank	Pearce	Taylor & Associates	9344 Kashiwa E		San Diego	CA
Mrs.	Dotty	Prenner	Cook & Sons	1900 S. Eads St	Suite	White Plains	NY
	Mike	Reynolds	Birch Catering	136 Harvey		San Francisco	CA
	Joseph	Schloss	Elmwood Insurance	11431 Williams		Fort Lee	NJ
	Robert	Sophie	Alvarado, Johnson, & Wr	3542 Roadside I		Berkeley	CA
Ms.	Charlene	Stein	Borregas-Wilson Inc.	250 Bellmarin E		Santa Ana	CA
Mrs.	Kathy	Abrams	Interplay Productions	750 Ridder Parl		Newport Beach	CA
	Clark	Alman	American Paint	81 Norwood Av		West Chester	PA
Ms.	Christy	Alpert	Signal Research	1120 Sharon Pa		Cupertino	CA

Using the `emptyfill` statement these empty cells can quickly be filled with `Mr.`

```
field T
emptyfill "Mr."
```

Here's the finished result.

T	First Name	Last Name	Company Name	Street Address	Suite Box	City	State
Mr.	Jim	Abrahms	International Transportat	329 North State		Alameda	CA
Ms.	Isabel	Alston	Leader Systems	10463 N. Blaney		San Rafael	CA
Mr.	Anthony	Campbell	Drake Inc.	3452 Van Ness		Chicago	IL
Dr.	Robert	Churchill	Smiley & Sons Developme	445 Hamilton		Manasquan	NJ
Ms.	Bea	Clairmont	Arrow Dev.	411 Pacific		Cambridge	MA
Mrs.	Dottie	Davidson	Van Ness Marketing	181 Taintor Dr.		Boston	MA
Mrs.	Barbara	Elliott	Foltene	10440 Torre Av		Cambridge	MA
Mr.	Lew	Farrell	Coast Label & Supply	33095 Lexington		San Rafael	CA
Ms.	Kris	Frazee	Mariani Publishing	18550 Lark Ave		San Francisco	CA
Mr.	Tim	Hill	Alameda Escrow	1000 Roche Blv		Santa Ana	CA
Mr.	Keith	Jacobs	Bath Co.	2994 Garcia Av		Burbank	CA
Mr.	Joe	Jones	Professionals Inc.	2 Owen St.		Provo	UT
Mr.	Al	Keizer	Certified Labs	1184 Lincoln Av	Suite	Westlake Village	CA
Ms.	Robin	Knight	Fico Appliance Service	2155 S. Bascom	Suite	Tallahassee	FL
Mr.	Mark	Lauing	Sherman-Davis	490 Broadway		Naples	FL
Dr.	Bill	Lieber	Arlington Associates	573 Dundee Rd.		Waldwick	NJ
Mr.	Russ	Malone	Northridge Bakeries	2210 Wilshire E	Suite	Chicago	IL
Ms.	Jan	Morgan	McCormick-Ridder	4044 Civic Terr		San Diego	CA
Mr.	Frank	Pearce	Taylor & Associates	9344 Kashiwa E		San Diego	CA
Mrs.	Dotty	Prenner	Cook & Sons	1900 S. Eads St	Suite	White Plains	NY
Mr.	Mike	Reynolds	Birch Catering	136 Harvey		San Francisco	CA
Mr.	Joseph	Schloss	Elmwood Insurance	11431 Williams		Fort Lee	NJ
Mr.	Robert	Sophie	Alvarado, Johnson, & Wr	3542 Roadside I		Berkeley	CA
Ms.	Charlene	Stein	Borregas-Wilson Inc.	250 Bellmarin E		Santa Ana	CA
Mrs.	Kathy	Abrams	Interplay Productions	750 Ridder Parl		Newport Beach	CA
Mr.	Clark	Alman	American Paint	81 Norwood Av		West Chester	PA
Ms.	Christy	Alpert	Signal Research	1120 Sharon Pa		Cupertino	CA

122 visible / 122 total

Automatic Numbering

The **sequence** statement fills the current field with a numeric sequence (for example 1, 2, 3 or 100, 110, 120). The **sequence** statement only works with numeric fields, you cannot sequence a text, date, or choice field. The sequence statement has one parameter, a text value that contains two numeric values within it, the starting value and the increment. Here's an example that will number 1000, 1001, 1002, etc.

```
field «Reg #»
sequence "1000 1"
```

Here is the result of this procedure.

Reg #	T	First Name	Last Name	Company	Street Address	Suite Box
1000	Mr.	Jim	Abrahms	International Transportat	329 North State	
1001	Ms.	Isabel	Alston	Leader Systems	10463 N. Blaney	
1002	Mr.	Anthony	Campbell	Drake Inc.	3452 Van Ness	
1003	Dr.	Robert	Churchill	Smiley & Sons Developme	445 Hamilton	
1004	Ms.	Bea	Clairmont	Arrow Dev.	411 Pacific	
1005	Mrs.	Dottie	Davidson	Van Ness Marketing	181 Taintor Dr.	
1006	Mrs.	Barbara	Elliott	Foltene	10440 Torre Av	
1007	Mr.	Lew	Farrell	Coast Label & Supply	33095 Lexington	
1008	Ms.	Kris	Frazee	Mariani Publishing	18550 Lark Ave	
1009	Mr.	Tim	Hill	Alameda Escrow	1000 Roche Blv	
1010	Mr.	Keith	Jacobs	Bath Co.	2994 Garcia Av	
1011	Mr.	Joe	Jones	Professionals Inc.	2 Owen St.	
1012	Mr.	Al	Keizer	Certified Labs	1184 Lincoln Av	Sui
1013	Ms.	Robin	Knight	Fico Appliance Service	2155 S. Bascom	Sui

The sequence can start with any number and increase by any value, including non-integer values or negative values. The table below shows four examples of starting and increment values.

"1 1"	"5 5"	"1 0.1"	"100 -1"
1	5	1.0	100
2	10	1.1	99
3	15	1.2	98
4	20	1.3	97
5	25	1.4	96

If the database contains summary records, the sequence count will reset to one after each summary record. If you want to sequence the current field without restarting at summary records, use the **formulafill** statement with the formula **seq()**. “[Summaries and Outlines](#)” on page 365 of the *Panorama Handbook* for more information on summary records. See “[Making Transformations Even Faster](#)” on page 573 for more information on the **formulafill** statement.

Propagate and UnPropagate

Like `emptyfill`, the `propagate` statement fills all the empty cells in the current field. However, instead of filling the empty cells with a fixed value, the `propagate` statement propagates filled data cells into the empty data cells (if any) below them. The `propagateup` statement propagates filled data cells into the empty data cells (if any) above them. See “[Propagate](#)” on page 466 of the *Panorama Handbook* for examples of these features in action.

The `unpropagate` statement performs the exact inverse of the `propagate` statement. If the same value appears in two or more consecutive data cells, the `unpropagate` statement empties the second and subsequent data cells. The `unpropagateup` statement performs the same operation upside down, leaving the last of several duplicate values while clearing the others. See “[UnPropagate](#)” on page 469 of the *Panorama Handbook* for examples of these features in action.

Using UnPropagate to Eliminate Duplicates

The `unpropagate` statement can be used to eliminate duplicate values in a database. To see how to do this manually see “[Using UnPropagate to Eliminate Duplicates](#)” on page 470 of the *Panorama Handbook*. This procedure will remove all of the duplicate entries in the current field.

```
sortup
unpropagate
select «» <> ""
removeunselected
```

Tip: One possible problem with this technique is that all cells that start out empty will be removed. For example if you are removing duplicate company names but some records don't contain company names, the records without company names will be removed. To fix this problem, use the `emptyfill` statement to fill the empty names with a unique value like `n/a` before you start, then use the `select` statement to select all values not equal (`≠` or `<>`) to `n/a`. Then perform the rest of the steps listed above. Here is a revised version of the procedure that takes care of this problem.

```
emptyfill "!empty!"
select «» <> "!empty!"
sortup
unpropagate
select «» <> ""
removeunselected
formulafill ?(«» = "!empty!" , "" , «»
```

Warning: Keep in mind that all of these techniques will blindly remove all but the first duplicate entry. In this example, there were two entries for [Bayshore Typesetting](#). However, they were probably not really duplicates, since one was in [Washington, DC](#) and the other in [San Rafael, CA](#). There is no way for an automatic technique like this to know which of these is really correct, or even if they are really duplicates at all. If you want to manually examine duplicate records instead of blindly deleting them, use the `selectduplicates` statement. See “[Selecting Duplicates](#)” on page 559 for more information on this statement.

Change (Find and Replace)

The `change` statement finds and replaces a word or phrase in the current field. For example, you can use the `change` statement to replace every occurrence of `Inc.` to `Incorporated`, or every occurrence of `Purchase Order` to `P.O.` In its most basic form the change statement has two parameters:

```
change <original text>,<new text>
```


To illustrate this statement we'll use this conference registration database. Notice that it contains the abbreviation **Inc.** in several places in the company name field.

T	First Name	Last Name	Company Name	Street Address	Suite Box	City	State
	Ms. Christy	Alpert	Signal Research	1120 Sharon Pa		Cupertino	CA
	Mr. Arthur	Clairmont	South Coast Office Produc	4390 Kaiser Dr		Cupertino	CA
	Mr. Harold	Cobb	Cobb Associates	3912 Phillip St.		Cupertino	CA
	Mrs. Sherry	Grossman	Pablo Distribution	1400 Valley Riv		Cupertino	CA
	Mr. Peter	Parks	Hamilton Press	41 Kenosia Ave		Cupertino	CA
	Mrs. Michelle	Adams	Sceptre	10159 Alliance		Cupertino	CA
	Mrs. Kathy	Schwartz	Wendover Insurance Grou	814 Castro St.		Long Beach	CA
	Mr. Charles	Arrow	Arrow, Inc.	390 Davis St.		Los Angeles	CA
	Mr. Dave	Elko	First Row Group	547 Jocom Way		Los Angeles	CA
	Mrs. Cindy	Blunden	Hot Lines, Inc.	#6 Hoover Pk		Palo Alto	CA
	Mr. Robert	Dorn	Valley Services	33 Cambridge P		Palo Alto	CA
	Mrs. Roxie	Jacobsen	Alpha Pic	174 Bellevue A		Palo Alto	CA

To change every occurrence of **Inc.** to **Incorporated** use this procedure.

```
field «Company Name»
change "Inc.", "Incorporated"
```

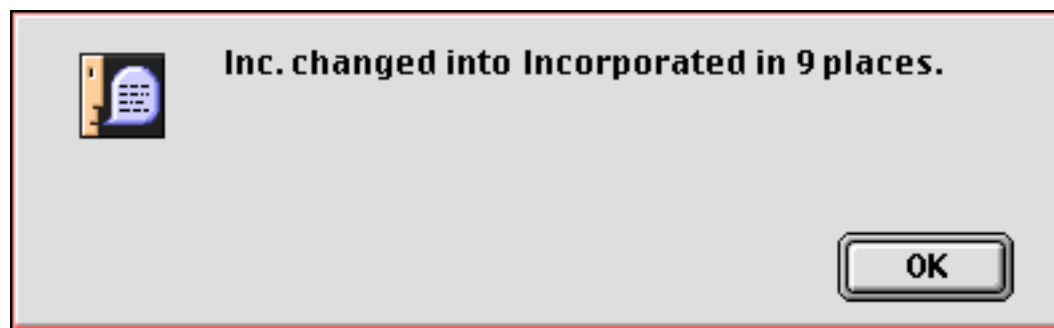
Running this procedure makes the changes.

T	First Name	Last Name	Company Name	Street Address	Suite Box	City	State
	Ms. Christy	Alpert	Signal Research	1120 Sharon Pa		Cupertino	CA
	Mr. Arthur	Clairmont	South Coast Office Produc	4390 Kaiser Dr		Cupertino	CA
	Mr. Harold	Cobb	Cobb Associates	3912 Phillip St.		Cupertino	CA
	Mrs. Sherry	Grossman	Pablo Distribution	1400 Valley Riv		Cupertino	CA
	Mr. Peter	Parks	Hamilton Press	41 Kenosia Ave		Cupertino	CA
	Mrs. Michelle	Adams	Sceptre	10159 Alliance		Cupertino	CA
	Mrs. Kathy	Schwartz	Wendover Insurance Grou	814 Castro St.		Long Beach	CA
	Mr. Charles	Arrow	Arrow, Incorporated	390 Davis St.		Los Angeles	CA
	Mr. Dave	Elko	First Row Group	547 Jocom Way		Los Angeles	CA
	Mrs. Cindy	Blunden	Hot Lines, Incorporated	#6 Hoover Pk		Palo Alto	CA
	Mr. Robert	Dorn	Valley Services	33 Cambridge P		Palo Alto	CA
	Mrs. Roxie	Jacobsen	Alpha Pic	174 Bellevue A		Palo Alto	CA

The procedure can use the `info("changecount")` function to find out how many occurrences of the word or phrase were changed (if any). Here is a modified version of the procedure that simply reports the number of changes.

```
field «Company Name»
change "Inc.", "Incorporated"
message "Inc. changed into Incorporated in "+str(info("changecount"))+ " places."
```

Running this revised procedure (on the original data) causes this message to appear.



By adding the **caps** option to the **change** statement you tell Panorama to adjust capitalization as it performs the replacement. The **caps** option should NOT be in quotes, and must be placed after the other parameters, separated by one or more spaces. For example:

```
field «Company Name»
change "Inc.", "Incorporated" caps
```

When the **caps** option is added to the statement Panorama will automatically adjust the capitalization of the new word or phrase as it is inserted into the database. If you leave this option off, capitalization is not adjusted. In fact, if the **caps** option is off, only words or phrases that exactly match the capitalization typed into the dialog will be replaced. The table below shows the result of replacing **Inc.** with **Incorporated** both with and without the **caps** option.

Original	without caps option	with caps option
Inc.	Incorporated	Incorporated
INC.	INC.	INCORPORATED
inc.	inc.	incorporated

By adding the **words** option to the **change** statement you tell Panorama to replace only entire words, not sections of words. For example, if you ask Panorama to change **is** to **was**, it will also change **this** to **thwas**. This is, of course, wrong. To prevent this, add the **words** option after the other parameters, like this.

```
field Body
change "is", "was" words
```

You can combine the **caps** and the **words** options, like this.

```
field Body
change "is", "was" caps words
```

The order of the **caps** and **words** options does not matter, as long as both are after the other parameters.

Changing with the Replace(Function

The `change` statement is not the only way to replace words or phrases. You can also use the `formulafill` statement and the `replace()` or `replacemultiple()` functions (see “[String Modification Functions](#)” on page 80). This technique is especially handy if you need to replace several words or phrases at once. For example, consider the addresses in the database below.

Company Name	Street Address	Suite Box	City
International Transportat	329 North State St.		Alameda
Pacific Micro	472 Wheelers Farms Rd.		Menlo Park
Interplay Productions	750 Ridder Park Dr.		Newport B
Minutemen Press	2150 Executive Dr.	Suite	San Mateo
Quantum Computer Servic	2082 Michelson Dr.	Suite	Vienna
Sceptre	10159 Alliance Rd.		Cupertino
Corporate Dynamics Inc.	1210 West Dayton St.		Redwood C
Hy-Ten	430 Clyde Avenue		Mountain V
Educational Resources	3431 Forest Circle		San Bruno
Kinetic Computing	1315 Bridgeway		Santa Ana
JPSA	3431 Forrest Bridge		Berkeley
American Paint	81 Norwood Ave.		West Ches
Signal Research	1120 Sharon Park Dr.		Cupertino
Leader Systems	10463 N. Blaney Ave.		San Rafael
Arrow, Inc.	390 Davis St.		Los Angele
ProLitho	215 Ace N		East Roche

Suppose you wanted to expand the abbreviations in these addresses: `St.` to `Street`, `Dr.` to `Drive`, etc. You could do this by using the `change` statement over and over again. Or you can simply use the `replacemultiple()` function to replace all of the abbreviations in one fell swoop.

```
field «Street Address»
formulafill replacemultiple(«Street Address»,
    "Rd./St./Dr./Ln./Ave.",
    "Road/Street/Drive/Lane/Avenue", "/" )
```

Running this procedure replaces all of the abbreviations at once.

Company Name	Street Address	Suite Box	City
International Transportat	329 North State Street		Alameda
Pacific Micro	472 Wheelers Farms Road		Menlo Park
Interplay Productions	750 Ridder Park Drive		Newport B
Minutemen Press	2150 Executive Drive	Suite	San Mateo
Quantum Computer Servic	2082 Michelson Drive	Suite	Vienna
Sceptre	10159 Alliance Road		Cupertino
Corporate Dynamics Inc.	1210 West Dayton Street		Redwood C
Hy-Ten	430 Clyde Avenue		Mountain V
Educational Resources	3431 Forest Circle		San Bruno
Kinetic Computing	1315 Bridgeway		Santa Ana
JPSA	3431 Forrest Bridge		Berkeley
American Paint	81 Norwood Avenue		West Ches
Signal Research	1120 Sharon Park Drive		Cupertino
Leader Systems	10463 N. Blaney Avenue		San Rafael
Arrow, Inc.	390 Davis Street		Los Angele
ProLitho	215 Ace N		East Roche

See “[Using FormulaFill to Transform Text](#)” on page 577 for more information on the `formulafill` statement.

Data Style and Color

In addition to the data stored in each cell, Panorama also keeps track of the style (plain, bold, italic, etc.) and (to a limited extent) color (red, green, etc.) of each cell (see “[Data Style and Color](#)” on page 474 of the *Panorama Handbook*). In a procedure the `stylecolor` statement can be used to change the style or color of one or more data cells. The statement has one parameter which controls what cells get changed (cell, record, field, all), what color the cells should be changed to (black, red, green, blue, cyan, yellow, magenta) and what style (bold, italic, underline, shadow).

If the parameter starts with the word `cell`, only the current cell will be changed. If the parameter starts with the word `record`, all the cells in the current record will be changed. If the parameter starts with the word `field`, all the selected cells in the current field will be changed. If the parameter starts with the word `all`, every cell in every selected record will be changed.

Here are some examples of different parameter combinations.

```
stylecolor "field blue bold"
stylecolor "all black"
stylecolor "cell red italic"
stylecolor "record bold"
```

For example, a procedure can underline the current data cell. We'll start with a plain data cell.

Hotel	City	Rate	Units	Phone	Star
General Palmer House	Durango	34.00	35	247-4747	4
Georgetown Motor Inn	Georgetown	24.50	32	569-3201	3
Glen Buff Motor Lodge	Boulder	30.00	75	442-7450	4
Glenwood Hot Springs Lodge	Glenwood Springs	32.00	71	945-6571	3
Greeley Lamplighter	Greeley	25.00	46	352-7070	3
Greenhorn Inn	Colorado City	30.00	59	279-2333	4
Greystone Guest Ranch	Evergreen	53.00	9	674-3328	2
Guest House Motel	Grand Junction	30.00	22	242-9571	3

Here is a procedure that changes the style of this cell.

```
stylecolor "cell underline"
```

When you run this procedure the cell will be underlined.

Hotel	City	Rate	Units	Phone	Star
General Palmer House	Durango	34.00	35	247-4747	4
Georgetown Motor Inn	Georgetown	24.50	32	569-3201	3
Glen Buff Motor Lodge	Boulder	30.00	75	442-7450	4
<u>Glenwood Hot Springs Lodge</u>	Glenwood Springs	32.00	71	945-6571	3
Greeley Lamplighter	Greeley	25.00	46	352-7070	3
Greenhorn Inn	Colorado City	30.00	59	279-2333	4
Greystone Guest Ranch	Evergreen	53.00	9	674-3328	2
Guest House Motel	Grand Junction	30.00	22	242-9571	3

It's easier to see the underline if you click on another cell.

Hotel	City	Rate	Units	Phone	Star
General Palmer House	Durango	34.00	35	247-4747	4
Georgetown Motor Inn	Georgetown	24.50	32	569-3201	3
Glen Buff Motor Lodge	Boulder	30.00	75	442-7450	4
Glenwood Hot Springs Lodge	Glenwood Springs	32.00	71	945-6571	3
Greeley Lamplighter	Greeley	25.00	46	352-7070	3
Greenhorn Inn	Colorado City	30.00	59	279-2333	4
Greystone Guest Ranch	Evergreen	53.00	9	674-3328	2
Guest House Motel	Grand Junction	30.00	22	242-9571	3

Using a slightly modified procedure we can make an entire line bold.

```
stylecolor "record bold"
```

Run the procedure to make the record bold.

Hotel	City	Rate	Units	Phone	Star
The Lodge At Georgetown	Georgetown	32.00	54	569-3211	3
The Lodge At Purgatory	Durango	40.00	50	247-9669	3
The Molly Gibson Lodge	Aspen	75.00	20	925-2580	4
The Nordic Lodge	Steamboat Springs	48.00	40	879-0531	3
The Raintree Inn	Colorado Springs	40.00	204	471-8680	4
The Stanley Sheraton Hotel	Estes Park	55.00	150	586-3371	3
The Swiss Chalet	Aspen	22.00	9	925-7146	2
The Victorian Inn	Telluride	60.00	20	728-3684	3

This procedure will make all phone numbers appear in italic blue, as shown here.

```
field Phone
stylecolor "field italic blue"
```

Here's the result.

Hotel	City	Rate	Units	Phone	Star
The Lodge At Georgetown	Georgetown	32.00	54	<i>569-3211</i>	3
The Lodge At Purgatory	Durango	40.00	50	<i>247-9669</i>	3
The Molly Gibson Lodge	Aspen	75.00	20	<i>925-2580</i>	4
The Nordic Lodge	Steamboat Springs	48.00	40	<i>879-0531</i>	3
The Raintree Inn	Colorado Springs	40.00	204	<i>471-8680</i>	4
The Stanley Sheraton Hotel	Estes Park	55.00	150	<i>586-3371</i>	3
The Swiss Chalet	Aspen	22.00	9	<i>925-7146</i>	2
The Victorian Inn	Telluride	60.00	20	<i>728-3684</i>	3

Notice that the italic blue has overridden the bold applied in the previous example.

For our final example we will go to a checkbook database and mark all insurance payments in green.

```
select Category="Insurance"
stylecolor "all green"
selectall
```

When you run this procedure all the insurance records will turn green, like this.

Date	CkNum	PayTo	Category	Debit
01/01/99		OPENING BALANCE		
01/08/99	1907	Northern Illinois Mold	Equipment Rental	96.05
01/08/99	1908	U S Postmaster	Postage	75.00
01/08/99	1909	Advertiser's Mailing Ser	Advertising	390.80
01/16/99	1910	Coudert Brothers, Attor	Legal Fees	223.52
01/16/99	1911	Paramount Stationers	Office Supplies	105.84
01/17/99	1912	California Capitol	Insurance	36.00
01/17/99	1913	California Capitol	Insurance	28.00
01/17/99	1914	U S Postmaster	Postage	75.00
01/17/99	1915	Sacramento Bee	Advertising	795.00
01/18/99		DEPOSIT		
01/22/99	1916	Walthers	Purchases	12,463.00
01/22/99	1917	Blue Cross Of Calif	Insurance	279.03
01/22/99	1918	Sherman Douglas Ins	Insurance	418.60
01/22/99	1919	Cannon Astro	Office Supplies	145.72
01/25/99	1920	Walthers	Purchases	1,885.40
01/25/99	1921	Nebs	Office Supplies	77.27
01/25/99	1922	Ramona Drinking Water	Office Supplies	98.10
01/25/99	1923	Pacific Partners	Rent	4,070.83
01/29/99	1924	Athearn	Purchases	1,906.32
01/29/99	1925	Advertiser's Mailing Ser	Advertising	860.22
01/29/99	1926	PacTel Cellular	Telephone	141.09
01/30/99	1927	State Board Of Equalizat	Taxes	549.00
01/30/99	1928	Walthers	Purchases	828.70
01/30/99	1929	Federal Express	Shipping	178.75
01/31/99	1930	U P S	Shipping	52.97
01/31/99	1931	Sacramento Bee	Advertising	795.00

A cell retains its style and color until the data is modified. Any data modification (editing, formula fill, etc.) will cause the cell to revert to plain black.

Every data cell that is not plain black takes up an extra byte of storage. For example a database with 10 fields and 500 records will expand by 5K bytes if you change every data cell to blue or italic (or both).

Accessing Style and Color in a Formula

Panorama formulas can use the `fieldstyle()` function to access both the style and color of individual data cells. When combined with the `select` statement, these functions allows you to select data based on its style or color. (See “[Selecting Information](#)” on page 557 for more information on this command.)

The basic syntax for the `fieldstyle()` function is:

```
fieldstyle(fieldname)
```

This function returns the style and color of a data cell— bold, italic, etc. The `fieldname` parameter is a string, so it should usually be in quotes—for example `fieldstyle("Price")="bold"`. If the data cell has more than one style or color, this function will return all of them, for example `red bold italic`. Use the `contains` operator (see “[String Testing Functions](#)” on page 78) to check for a specific style or color, for example

```
select fieldstyle("Name") contains "italic"
```

To check if a cell is plain, use a formula like this

```
fieldstyle("Address")=""
```

For more information on this function see “[FIELDSTYLE\(\)](#)” on page 5221 of the *Panorama Reference*.

Processing/Transforming an Entire Array

The previous section described methods for transforming an entire field in a database (See “[Transforming Big Chunks of Data](#)” on page 572). This section describes different methods for quickly processing all the elements in an array. If you are not already familiar with text arrays see “[Text Arrays](#)” on page 93.

“Filtering” an Array

The `arrayfilter` statement allows a procedure to use a formula to process each element of an array. The statement scans the array you specify element by element, and uses the formula you supply to transform each element. It then builds a new array using the transformed elements.

The `arrayfilter` statement has four parameters:

```
arrayfilter oldarray,newarray,separator,formula
```

The first parameter, `oldarray`, is the original array. This is usually a field or variable, but could be any formula that produces text data.

The second parameter, `newarray`, is the new array. This must be a field or variable. If you don't mind changing the original array, `oldarray` and `newarray` may be the same field or variable. If you want to keep the original array, `newarray` should be different.

The third parameter is the `separator` character (see “[Picking a Separator Character](#)” on page 93).

The fourth parameter is the `formula` that will transform each array element. In addition to the usual functions and operators there are two functions that have special meaning in this function. The `import()` function returns the original data in the array element. The `seq()` function returns the array element number (1, 2, 3, etc.).

Here is a procedure called `.NumberArray` that adds sequence numbers to an array. The procedure that calls `.NumberArray` must pass an array as parameter 1 and a separator character as parameter 2.

```
local tempArray
tempArray=parameter(1)
arrayfilter tempArray,tempArray,parameter(2),(" "+str(seq())) "+import()
setparameter 2,tempArray
```

Here's a procedure that uses `.NumberArray` to produce a numbered list of atomic elements.

```
local Elements
Elements=replace("Hydrogen;Helium;Lithium;Beryllium;Boron;"+
  "Carbon;Nitrogen;Oxygen;Flourine;Neon;"+
  ...
  "Mendelevium;Nobelium;Lawrencium",";","")
.call .NumberArray,Elements,¶
```

This table shows what the **Elements** array looks like before and after the **.NumberArray** procedure processes the array.

Before	After
Hydrogen	(1) Hydrogen
Helium	(2) Helium
Lithium	(3) Lithium
Beryllium	(4) Beryllium
Boron	(5) Boron
Carbon	(6) Carbon
Nitrogen	(7) Nitrogen
Oxygen	(8) Oxygen
Fluorine	(9) Fluorine
Neon	(10) Neon
...	...
...	...
Mendelevium	(101) Mendelevium
Nobelium	(102) Nobelium
Lawrencium	(103) Lawrencium

Stripping Blank Elements From An Array

The **arraystrip()** function removes all blank elements from an array. This function has two parameters: the original array, and the separator character for that array.

```
arraystrip(array,separator)
```

The **arraystrip()** function can be combined with the **arrayfilter** statement to produce a subset of an array. This example creates a list of recent local phone numbers.

```
global RecentLocalPhone
arrayfilter RecentPhone,RecentLocalPhone,¶,
    ?(length(import())<10,import(),"")
RecentLocalPhone=arraystrip(RecentLocalPhone,¶)
```

The table below shows how this procedure works. The first column shows the original **RecentPhone** array, with 6 phone numbers, 3 local, 3 in other area codes. The second column shows the **RecentLocalPhone** array after the **arrayfilter** statement. Using the **?(** function the formula has “blanked out” all the phone numbers with area codes (see “[The ? Function](#)” on page 130). The array elements are still there, but they are empty. The third column shows the **RecentLocalPhone** array after the **arraystrip()** function. All the empty array elements have been removed, so this array now has only 3 items.

Original	after ArrayFilter	after ArrayStrip(
784-3490	784-3490	784-3490
(213) 454-3309		940-2234
(408) 339-7792		878-2256
940-2234	940-2234	
(303) 452-2284		
878-2256	878-2256	

Reversing the Order of an Array

The **arrayreverse()** function reverses the order of the elements of an array. For example, the formula:

```
arrayreverse("1;2;3;4",";")
```


will produce the array 4;3;2;1.

The formula below could be used with an auto-wrap or text display object to display all the checks written to a company, starting with the most recent check (assuming the Checkbook database is sorted by date).

```
arrayreverse(lookupall("Checkbook",«Pay To»,Company,«Check Num»,¶),¶)
```

Using Regular Text Functions with Arrays

Don't forget that you can use regular text processing techniques on arrays (see "Text Formulas" on page 67). After all, an array is simply a chunk of text that happens to have separator characters in it. For example, to convert our entire atomic element array to upper case in one fell swoop, just use this assignment:

```
Elements=upper(Elements)
```

The assignment below will change all the stainless steel parts in the `PartsList` array to cheap plastic.

```
PartsList=replace(PartsList,"Stainless Steel","Cheap Plastic")
```

Of course no Panorama customer will ever need a formula like that!

Sorting an Array

Sorting an array in a procedure is easy. The `arraysort` statement does all the work. This statement has three parameters:

```
arraysort oldarray,newarray,separator
```

The first parameter, `oldarray`, is the original array. This is usually a field or variable, but could be any formula that produces text data.

The second parameter, `newarray`, is the new array. This must be a field or variable. If you don't mind changing the original array, `oldarray` and `newarray` may be the same field or variable. If you want to keep the original array, `newarray` should be different.

The third parameter is the `separator` character.

The `arraysort` statement always sorts the array in ascending order (A's first, Z's last). Upper case, lower case, and international letters will be sorted correctly (i.e. `a` comes before `B`, which may seem obvious but is not the normal ASCII order).

The example below builds a fileglobal variable named `FormList` that contains a list of the forms in the current database. The list is carriage return delimited and alphabetized. You could use this list with a Pop-Up Menu or a List SuperObject™.

```
fileglobal FormList
FormList=dbinfo("forms","")
arraysort FormList,FormList,¶
```

Removing Duplicate Items from an Array

The `arraydeduplicate` statement also sorts an array. After it sorts the array it eliminates all the elements that are duplicated. For example, if an array contains `San Francisco` three times, this statement will eliminate two and leave only one. The parameters for the `arraydeduplicate` statement are the same as for the `arraysort` statement.

```
arraydeduplicate oldarray,newarray,separator
```

The example below creates a fileglobal variable named **Companies**, then fills it with a sorted list that contains all the companies in **California** listed in alphabetical order:

```
fileglobal Companies
Companies=lookupall("Invoices",State,"CA",Company,¶)
arraydeduplicate Companies,Companies,¶
```

Each company will be listed only once, no matter how many times the company appears in the **Invoices** database.

(Note: There is no automatic way to eliminate the duplicate values in an array without also sorting the array.)

Building an Array from a Database

The previous example showed how an array can be built from the data in a database using the **lookupall()** function. An even more powerful technique is the **arraybuild** statement. This statement scans a database and, using a formula you supply, extracts information from each record to build an array. The statement has four parameters:

```
arraybuild array,separator,database,formula
```

The first parameter, **array**, is the new array you want to build. This must be a field or variable.

The second parameter is the **separator** character (see “[Picking a Separator Character](#)” on page 93).

The third parameter, **database**, is the name of the database that contains the information that will be scanned to build the array. This database must be currently open at the time the **arraybuild** statement is used. If you want to use the current database use the function **info("databasename")** or simply use **"**.

The fourth parameter, **formula**, is the heart of this statement. As the **arraybuild** statement scans the database record by record, it uses this formula to extract data from each record and add it to the array. The formula can also be used to select which records appear in the array. If the formula produces empty text for a particular record, that record will not be included in the array. The formula can reference any field in the database being scanned.

The example below produces a list of past due invoices.

```
fileglobal PastDueAccounts
arraybuild PastDueAccounts,¶,"Invoices",
    ?(Balance>0 and today()-InvoiceDate>30,
    pattern(InvoiceNumber,#####)+" (" +Company+" )", "")
```

This procedure will produce an array that will look something like this:

```
00436 (Acme Widgets)
02445 (Optimal Resolution Trust)
03689 (Zippy Car Wash)
```

This array could be displayed on a report, or it could be used in a pop-up menu or scrolling list.

Warning: One thing to be careful about when building arrays with the **arraybuild** statement is the size (number of characters) of the array you are building. The array must fit in Panorama’s scratch memory allocation.

Note: The **arraybuild** statement scans every record in the database, whether it is selected or not. If you would only like to scan selected records, use the **arrayselectedbuild** statement. This statement is identical to the **arraybuild** statement except for the fact that it only scans selected records. If you know that all the records you want to scan are selected, this statement may be much faster than the regular **arraybuild** statement.

Appending an Array to a Database

An array in a variable can be appended to a database almost as if it were a text file on the disk. The array must be carriage return delimited, and if there is more than one field, the fields must be tab delimited. To append a variable to the current database, put `+`@ in front of the variable name like this:

```
openfile "+@Array"
```

To replace the entire current database with the array, put `&`@ in front of the variable name like this:

```
openfile "&@Array"
```

Here is an example that transfers houses from a [Listings](#) database to a [Sales](#) database.

```
local TransferArray
arraybuild TransferArray,␣,"Listings",
    ?(Escrow≠"Closed", "",
    Date+--+Address+--+City+--+State+--+Zip+--+str(SoldPrice) )
select Escrow≠"Closed"
removeunselected
open "Sales"
open "+@TransferArray"
```

The `arraybuild` statement copies all the recently sold listings into the `TransferArray` variable. The `-` character (see “[Special Characters](#)” on page 57) separates each field with the required tab. Once the listings are safely copied into the array, they are deleted from the [Listings](#) database. The first `open` statement makes sure that the [Sales](#) database is open and on top. This database has six fields, [SoldDate](#), [Address](#), [City](#), [State](#), and [SoldPrice](#), in that order. The formula in the `arraybuild` statement has been set up to create the array with the fields in that order. The final statement of the procedure appends the data in `TransferArray` to the end of the [Sales](#) database.

If you are transferring information between two databases with identical fields the `exportline()` function can come in handy. This function outputs all the fields in the current record with tabs in between. The example below appends all of the invoices in the database [Paul's Invoices](#) to the current database.

```
local TransferArray
arraybuild TransferArray,␣,"Paul's Invoices",exportline()
openfile "+@TransferArray"
```

This example transferred all of the information across, but you could use the `?(` function to transfer just a subset.

Copying Between Multiple Variables and an Array

Panorama has the ability to combine multiple variables into a single array, or to take an array and split it into many separate variables. This capability can be useful for editing arrays (each array element can be edited in a separate variable) and for saving a collection of variables in a single disk file (for example to store preferences).

The `savevariables` statement takes a list of variables and combines the values of all the variables into a single array. The statement has three parameters:

```
savevariables VariableList,CombinedArray,Separator
```

The `VariableList` parameter is an array containing the names of the variables to be included in the result. Each item in the array must be separated from the next by the `Separator` character (see “[Picking a Separator Character](#)” on page 93).

The **CombinedArray** parameter is a field or variable name. The statement will build the final array of values in this field or variable, using the **Separator** character to divide each item. Any numbers will be converted to text as the array built. The example below saves all of the **fileglobal** variables for the current file into a disk file.

```
local fileExtraData
savevariables info("filevariables"),fileExtraData,¶
filesave "",info("databasename")+ " Variables", "",fileExtraData
```

The following example is similar but it saves both the variable name and the data in the format **variable=value**. The variables will be listed in alphabetical order.

```
local varNames,varData
varNames=info("filevariables")
arraysort varNames,varNames,¶
savevariables info("filevariables"),varData,¶
arrayfilter varData,varData,¶,array(info("filevariables"),seq(),¶)+"="+import()
filesave "",info("databasename")+ " Variables", "",varData
```

The resulting file will be named something like **Contact Variables** (the exact name depends on the database name) and will look something like this:

```
ActiveForm=Contacts
LocalAreaCode=714
SearchText=Chicago
```

The **loadvariables** statement takes an array of values and splits the values into individual variables. If the variables don't exist, the statement will create them. The **loadvariables** statement has three parameters:

```
loadvariables VariableList,VariableValues,Separator
```

The **VariableList** parameter is an array containing the names of the variables to be loaded. Each item in the array must be separated from the next by the **Separator** character (see "[Picking a Separator Character](#)" on page 93).

The **VariableValues** parameter is an array containing the values of the variables. This array must be in the same order as the **VariableList** parameter. Each value in the array must be separated from the next by the **Separator** character.

Here is an example that loads three variables from an array.

```
loadvariables "Gold,Silver,Bronze","Johnson,Smith,Fetzl",",","
```

This example is exactly the same as:

```
Gold="Johnson"
Silver="Smith"
Bronze="Fetzl"
```

If a variable already contains a numeric value, the **loadvariables** statement keeps it numeric if possible (if all the characters in the new value are numeric). Here is an example where three numbers are loaded into variables.

```
fileglobal Red,Green,Blue
Red=0 Green=0 Blue=0
loadvariables "Red,Green,Blue","24,58,199",",","
```

This example is exactly the same as the procedure below. Notice that there are no quotes around the numbers.

```
fileglobal Red,Green,Blue
Red=24
Green=58
Blue=199
```

So far the examples aren't too exciting. Here is an example that is a bit more interesting. Suppose you had an array called **ContactInfo** that contained name/value pairs like this:

```
Name:Johnson
Email:ajohnson@worldwide.com
url:www.worldwide.com
```

The example below can take this array and separate it into three variables called **ContactName**, **ContactEmail**, and **Contacturl**.

```
global ContactInfo
local contactVariables,contactValues
arrayfilter ContactInfo,contactVariables,¶,"Contact"+array(import(),1,":")
arrayfilter ContactInfo,contactValues,¶,array(import(),2,":")
loadvariables contactVariables,contactValues,¶
```

The procedure starts by splitting the **ContactInfo** array into two separate arrays for variable names and values, then creates and loads the variables with the values.

The **loadvariables** statement will automatically create fileglobal variables if they do not exist. If you want to create some other kind of variable you can use one of the four statements listed below. (The **loadfileglobalvariables** statement is actually exactly the same as the **loadvariables** statement, but is included for completeness.)

```
LoadGlobalVariables VariableList,VariableValues,Separator
LoadFileGlobalVariables VariableList,VariableValues,Separator
LoadWindowVariables VariableList,VariableValues,Separator
LoadLocalVariables VariableList,VariableValues,Separator
```

If the variables have already been created then these four statements all work exactly the same.

Editing an Array using Separate Variables

The **loadvariables** and **savevariables** statements can be used to help edit an array as individual components. Each component has its own SuperObject for text editing, but the results are all combined into a single variable.

Start by defining the variable names for each individual line item component. You'll also need to build a form with SuperObjects to edit each of these variables.

```
fileglobal LineItems
LineItems=replace("Qty1/Desc1/Price1/Qty2/Desc2/Price2","/",¶)
```

When you open the form to edit the array, this procedure will fill the separate variables with data from the **LineItemData** array (which could be a field or a variable).

```
loadvariables LineItems,LineItemData,¶
```

When a component is modified you can rebuild the combined array with this procedure.

```
savevariables LineItems,LineItemData,¶
```

Processing/Transforming Binary Data

By now probably everyone who has ever used a computer for more than a week has heard that at their core, computers work with 1's and 0's, on and off, true and false. This is called binary data, because there are only two options. Fortunately, users don't ever have to deal with raw binary data. The programmers take the 1's and 0's and give them structure to create text, numbers, pictures, and other complex elements.

It's not much fun, and it's rarely necessary, but Panorama does allow a procedure programmer to work with raw, unstructured, binary data: 1's and 0's. When you work with raw binary data it will always be in a text field or variable. Panorama normally interprets text as a series of characters, as described earlier in this chapter. The binary functions, however, do not interpret the binary data as characters. Instead, they allow you to directly access and manipulate the 1's and 0's. Panorama uses the text data type to hold raw binary data because text may be of any length and places no restrictions on the binary information that is placed in it. (However, the text may look very strange if you display it in the data sheet or on a form; more on this later.)

Bits

The fundamental unit of computer information is a bit. A bit contains a single 1 or 0. However, a bit is too small to be of much use by itself, so usually several bits are grouped together into a collection called a byte, word, or longword.

Function	Reference Page	Description
bit(number)		This function converts a bit number (1 to 32) into a number (1, 2, 4, 8, 16, etc.)
getbit(number,bitnumber)		This function returns a true or false value by testing a bit. The bit number may be from 1 to 32. If the bit is set, the value will be true, if it is not set, the value will be false.
onescomplement(number)		This function returns the one's complement of a 32 bit number (all bits are reversed)
setbit(number,bitnumber,truefalse)		This function sets one bit within a number, without disturbing any of the other bits. The bit number may be from 1 to 32. The bit will be set based on the true/false parameter - set if true, cleared if false.

Bytes

A byte is a collection of 8 bits. There are 256 possible combinations of 1's and 0's within a byte (2^8). These 256 possible combinations could represent characters, they could represent numbers from 0-255 or they could represent 256 of anything else. The byte(function takes a number from 0 to 255 and converts it into the corresponding pattern of 8 bits.

Function	Reference Page	Description
byte(number)	Page 5079	This function converts a number into a single byte of binary data. (Note: the byte(function is basically the same as the chr(function.) The number parameter must be between 0 and 255. This function converts the number into a single byte of binary data (8 bits).
binaryvalue(data)	Page 5075	This function converts binary data (a byte, word, or longword) into a number. Data is the binary value that you want to convert into a number. This value must be a byte, a word (2 bytes) or a longword (4 bytes). The result is an integer.
bytearray(data,index)		This function extracts a value from an array of bytes. This is not a Panorama style delimited array but a C style array of 8 bit values. The result is an integer.

Function	Reference Page	Description
<code>chararray(data,index)</code>		This function extracts a characters from an array of characters. This is not a Panorama style delimited array but a C style array of ASCII characters. The result is an single characters.

Words

A word is a collection of 16 bits (or 2 bytes). There are 65,536 possible combinations of 1's and 0's within a word (2¹⁶). These 65,536 possible combinations could represent numbers from 0-65,535 or they could represent 65,536 of anything else. The `word()` function takes a number from 0 to 65,535 and converts it into the corresponding pattern of 16 bits.

Function	Reference Page	Description
<code>word(number)</code>	Page 5906	This function converts a number into a single word (2 bytes) of binary data. Number is the value that you want to convert into a binary number. This value must be between 0 and 65,535. This function converts the number into a two bytes of binary data (16 bits).
<code>binaryvalue(data)</code>	Page 5075	This function converts binary data (a byte, word, or longword) into a number. Data is the binary value that you want to convert into a number. This value must be a byte, a word (2 bytes) or a longword (4 bytes). The result is an integer.
<code>wordarray(data,index)</code>		This function extracts a value from an array of words. This is not a Panorama style delimited array but a C style array of 16 bit values. The result is an integer.

Long Words

A longword is a collection of 32 bits (or 4 bytes). There are over 4 billion possible combinations of 1's and 0's within a longword (2³²). The `longword()` function takes a number from 0 to 4,294,967,295 and converts it into the corresponding pattern of 32 bits.

Function	Reference Page	Description
<code>longword(number)</code>	Page 5499	This function converts a number into a single longword (4 bytes) of binary data. Number is the value that you want to convert into a binary number. This value must be an integer. This function converts the number into a four bytes of binary data (32 bits).
<code>binaryvalue(data)</code>	Page 5075	This function converts binary data (a byte, word, or longword) into a number. Data is the binary value that you want to convert into a number. This value must be a byte, a word (2 bytes) or a longword (4 bytes). The result is an integer.
<code>longwordarray(data,index)</code>		This function extracts a value from an array of longwords. This is not a Panorama style delimited array but a C style array of 32 bit values. The result is an integer.

Creating Binary Values

Binary values are created with the `byte()`, `word()` and `longword()` functions. The example below builds a text data value from a longword, a word, a word and a byte. The resulting text item has a length of 9 (4+2+2+1).

```
global rawData
rawData=longword(96345)+
  word( 1249)+
  word( 9004)+
  byte( 80)
```

Numeric values can be recovered from a text data item with the `binaryvalue()` function. The text input into this function must have a length of 1, 2, or 4. You can use text funnels to control the position and length of the data being converted. The example below will extract four values from a text item that is at least 9 bytes long.

```
global rawData
local myLong,myFirstWord,mySecondWord,myByte
myLong= binaryvalue( rawData[1;4])
myFirstWord= binaryvalue( rawData[5;2])
myFirstWord= binaryvalue( rawData[7;2])
myByte= binaryvalue( rawData[9;1])
```

If `rawData` contains the information stored in it from the previous example, the `myLong` variable will contain **96345**, `myFirstWord` will be **1249**, `mySecondWord` will be **9004**, and `myByte` will be **80**.

One Dimensional Arrays of Binary Values

Programmers familiar with languages like C and Pascal will be familiar with the concept of an array as a sequence of binary values, for example a sequence of bytes, words, longwords, or some other length of binary data. This is quite different from Panorama's usual concept of arrays as variable length items separated by a special character. Panorama has several functions for extracting a specific element from an array of binary values. The function you pick depends on the type of binary value in the array.

Function	Reference Page	Description
<code>bytearray(data,index)</code>		This function extracts a value from an array of bytes. This is not a Panorama style delimited array but a C style array of 8 bit values. The result is an integer.
<code>chararray(data,index)</code>		This function extracts a characters from an array of characters. This is not a Panorama style delimited array but a C style array of ASCII characters. The result is an single characters.
<code>chunkarray(data,index, chunklength)</code>		This function extracts a binary chunk from an array of chunks. This is not a Panorama style delimited array but a C style array of binary chunks. The result is a binary value (text).
<code>longwordarray(data,index)</code>		This function extracts a value from an array of longwords. This is not a Panorama style delimited array but a C style array of 32 bit values. The result is an integer.
<code>wordarray(data,index)</code>		This function extracts a value from an array of words. This is not a Panorama style delimited array but a C style array of 16 bit values. The result is an integer.

Panorama also has three statements designed for rapidly filtering and/or processing arrays of binary data — `characterfilter`, `chunkfilter`, and `textfilter`.

The CharacterFilter Statement

This statement scans text on a character by character basis. As it scans the input text, it builds a new text field or variable. The contents of the new text is based on the result of the formula which is applied to each character of the original text. The formula can use the `import()` function to retrieve the original character, or the `seq()` function to retrieve the position of the character within the text.

The `CharacterFilter` statement has three parameters: `OriginalText`, `NewText` and `Formula`.

The `OriginalText` parameter is the text you want to scan. This may be a field, a variable, or any formula that calculates a text value.

The `NewText` parameter is the name of a field or variable for holding the output text. If this is a field, it must be a text field.

The **Formula** parameter is a formula that produces a text result. This formula will be calculated for each character in the **OriginalText** parameter. The result will be appended to the **NewText** field or variable. The result of this formula isn't limited to a single character, but may be several characters or zero characters (""). Within the formula you can use the **import()** function to retrieve the original character, or the **seq()** function to retrieve the position of the character within the text.

Let's review some examples that illustrate the operation of the **CharacterFilter** statement. This example converts a string of characters into a comma separated array. For example, if A is **abcd**, then B will become **a,b,c,d, .**

```
characterfilter A,B,import()+", "
```

This example reverses the characters in a string of text. For example, if A is **abcd**, then B will become **dcba .**

```
characterfilter A,B,A[-seq();1]
```

This example converts text into a set of hex digits. For example, if A is **abcd**, then B will become **61 62 63 64 .**

```
characterfilter A,B,radixstr("hex",asc(import()))[-2,-1]+" "
```

The **ChunkFilter** Statement

This statement is almost the same as **CharacterFilter**, but instead of single characters it allows you to process fixed length chunks. Useful for base64 encryption, hex/ascii conversion, cryptography, etc. As it scans the input text, it builds a new text field or variable. The contents of the new text is based on the result of the formula which is applied to each character of the original text. The formula can use the **import()** function to retrieve the original chunks of characters, or the **seq()** function to retrieve the chunk number (first chunk is 1, next is 2, etc..).

The **ChunkFilter** statement has four parameters: **ChunkSize**, **OriginalText**, **NewText** and **Formula**.

The **ChunkSize** parameter is the size of the chunks you want to use. This may be a field, a variable, or any formula that calculates a numeric (integer) value, for example 2, 3, 5 or 10.

The **OriginalText** parameter is the text you want to scan. This may be a field, a variable, or any formula that calculates a text value.

The **NewText** parameter is the name of a field or variable for holding the output text. If this is a field, it must be a text field.

The **Formula** parameter is a formula that produces a text result. This formula will be calculated for each character in the **OriginalText** parameter. The result will be appended to the **NewText** field or variable. The result of this formula isn't limited to the same size as the original chunk, but may be several characters or zero characters (""). Within the formula you can use the **import()** function to retrieve the original character, or the **seq()** function to retrieve the position of the character within the text.

To see examples of the **chunkfilter** statement in a real application, check out the **BASESIXTYFOURENCODE** and **BASESIXTYFOURDECODE** procedures in the **_InternetLib** custom statement library.

The **TextFilter** Statement

This statement scans text and creates new text based on the original text. Unlike the **CharacterFilter** statement, the **TextFilter** statement doesn't necessarily scan the text character by character. Instead, it allows you to skip over a sequence of characters based on a formula before it starts scanning character by character again.

The **TextFilter** statement starts with the first character in the original text. It then calculates the result of a formula based on this location. This formula must start with the **longword()** function. The number in this function determines the number of characters to skip before continuing the scan. If this skip value is zero, the **TextFilter** statement stops scanning. If there is any text after the **longword()** value, that text is appended to the output text.

The **TextFilter** statement has three parameters: **OriginalText**, **NewText** and **Formula**.

The **OriginalText** parameter is the text you want to scan. This may be a field, a variable, or any formula that calculates a text value.

The **NewText** parameter is the name of a field or variable for holding the output text. If this is a field, it must be a text field.

The **Formula** parameter is a formula that produces a text result. The first four bytes of the result from this formula is treated as a binary value that tells Panorama how many characters to skip before resuming scanning (you can produce this binary value with the `longword()` function). Any additional text after this binary value will be appended to the **NewText** field or variable. Within the formula you can use the `import()` function to retrieve the original text starting from the current location, or the `seq()` function to retrieve the position of the character within the text.

Let's review an example that illustrate the operation of the **TextFilter** statement. This example take some text in A and then place in B a space separated array containing the word lengths. For example, if A contains

How long is each word in this sentence?

Then the result in B will be

3 4 2 4 4 2 4 9

Here is the code that performs this job:

```
textfilter A,B,
  longword(1+length(array(import(),1," "))) +
  str(length(array(import(),1," "))) + ", "
```

There's just one problem with this simple example - there's an even simpler way to do this that doesn't use the **TextFilter** statement:

```
arrayfilter A,B," ",str(length(import()))
```

To see examples of the **chunkfilter** statement in a real application, check out the **FINDALLINTEXT**, **HARDWRAP** and **HEXDUMP** procedures in the `_TextLib` custom statement library.

Data Dictionaries

A conventional dictionary (Webster's, for example) contains a list of entries. Each entry comes as a pair — the word itself, and the definition of the word.

Panorama supports a data structure called a **data dictionary**. Like a conventional dictionary, a data dictionary contains a list of entries, and each entry comes as a pair — the entry **key** (which identifies the entry) and the entry **value** (some data associated with this entry). Sometimes these entries are referred to as **key/value pairs**. If you know the key, you can find out what the value is. A data dictionary allows you to combine any number of these key/value pairs into a single structure that can be stored in a single variable, a single field or a single procedure parameter. For example, in a database you could use a data dictionary to store additional, seldom used data that you don't want to devote an entire field to, or to store information that you didn't anticipate when you created the database. When a procedure (or custom statement) has a large and variable number of parameters, you can combine these into a data dictionary that can be passed back and forth easily (many of Panorama's Internet access statements work this way). Data dictionaries are also an excellent way to store preferences, and they are used that way by many of Panorama's wizards (for example the **Hot Key** wizard and any application that works with Generic fields (see "[Generic Fields](#)" on page 230 of the *Panorama Handbook*.)

Key/Value Pairs

Each entry in a dictionary consists of a key/value pair. The key identifies the pair, and must be unique. In other words in a single dictionary you cannot have two pairs with the same key — every pair must have its own unique name. The key can contain any text character you want, including spaces and punctuation. (The only characters that are not allowed are `chr(254)` and `chr(255)`.)

The value is simply a text value that is associated with the key. The value may contain any character except for `chr(254)` and `chr(255)`. Here are some examples of typical key/value pairs.

Key	Value
Color	Red
Area Code	951
Greeting	Dear
Home Page	www.myhomepage.com

These examples are for illustration purposes only — you can use just about anything you need as a key or value.

Storing a Key/Value Pair in a Dictionary

To store a key/value pair in a dictionary use the `setdictionaryvalue` statement.

```
setdictionaryvalue dictionary,key,value
```

The `dictionary` parameter tells the statement where the dictionary is located. This must be the name of a field or variable. If a variable, the variable must be defined beforehand with the `local`, `fileglobal`, or `global` statements (see “[Variables](#)” on page 247). The variable must also have a defined value, typically `""` for a brand new dictionary.

The `key` parameter is the identifier for this key/value pair. The `value` parameter is the actual data that will be stored. If this key already exists within the dictionary the original value will be replaced by the new value, otherwise a new key/value pair will be added to the dictionary.

To illustrate, here is an example that creates a new dictionary variable name `ColorPalette`, then fills it with three key/value pairs.

```
local ColorPalette
ColorPalette=""
setdictionaryvalue ColorPalette,"TextColor","red"
setdictionaryvalue ColorPalette,"ButtonColor","blue"
setdictionaryvalue ColorPalette,"Background","white"
```

Now the `ColorPalette` variable contains all three key/value pairs — `TextColor/red`, `ButtonColor/blue` and `Background/white`.

You can change the value of each key/value pair individually at any time. This line of code will change the `ButtonColor` to `green`.

```
setdictionaryvalue ColorPalette,"ButtonColor","green"
```

If necessary, you can also delete individual key/value pairs with the `deletedictionaryvalue` statement.

```
deletedictionaryvalue ColorPalette,"ButtonColor"
```

This statement completely removes the `ButtonColor` key/value pair from the `ColorPalette` dictionary.

Accessing Dictionary Entries

Once values have been stored in a dictionary they can be retrieved with the `getdictionaryvalue()` function, like this.

```
theColor=getdictionaryvalue(Palette,"TextColor")
```

The first parameter is the name of the field or variable that contains the dictionary, while the second parameter is the key value of the item you want to retrieve.

If you are not sure if a dictionary contains a particular key/value pair you can use the `dictionaryvalueexists()` function to find out. This example checks to see if a button color has been defined, and if so calls a subroutine to change the color of all buttons.

```
if dictionaryvalueexists(Palette,"ButtonColor")
    call ChangeAllButtonColor,getdictionaryvalue(Palette,"ButtonColor")
endif
```

For a complete list of all of the entries in the dictionary use the `listdictionarynames()` function. This function has one parameter, the name of the field or variable that contains the dictionary.

```
message listdictionarynames(ColorPalette)
```

The output is a text array that lists the keys stored in the dictionary, with each item separated by a carriage return.



The `dumpdictionary` statement outputs a carriage return delimited array that contains both the keys and the values, separated by an equals sign.

```
local ColorPalette,PaletteDump
dumpdictionary ColorPalette,PaletteDump
message PaletteDump
```

The output displayed by the message statement will look something like this:



The `dumpdictionaryquoted` statement is similar, but each key is surrounded by chevrons and each value is surrounded by quotes.

```
local ColorPalette,PaletteDump
dumpdictionaryquoted ColorPalette,PaletteDump
message PaletteDump
```

Here's the output:



If you have fields or variables with the same names as the keys in your database, you can use this output with the `execute` statement to initialize all of the fields or variables at once.

```
local ColorPalette,PaletteDump
dumpdictionaryquoted ColorPalette,PaletteDump
execute PaletteDump
```

Another Technique For Initializing a Dictionary

Earlier we showed you a method for setting up and initializing a dictionary.

```
local ColorPalette
ColorPalette=""
setdictionaryvalue ColorPalette,"TextColor","red"
setdictionaryvalue ColorPalette,"ButtonColor","blue"
setdictionaryvalue ColorPalette,"Background","white"
```

Panorama also includes a special statement for this task, `initializedictionary`. Here is the same task performed with this statement.

```
local ColorPalette
initializedictionary ColorPalette,
  "TextColor","red",
  "ButtonColor","blue",
  "Background","white"
```

You can include as many key/value pairs as you need. Using this statement instead of separate `setdictionaryvalue` statements has two advantages — it takes less typing, and it runs faster. Of course you can always modify individual key/value pairs later with the `setdictionaryvalue` statement.

Additional Methods for Modifying Dictionary Entries

The `appenddictionaryvalue` statement appends additional text to the value portion of an existing key/value pair. For example, the statement below will add `75%` to whatever the `ButtonColor` is, so `Blue` becomes `Blue 75%`. (The third parameter is a separator character, which in this case is a space.)

```
appenddictionaryvalue ColorPalette,"ButtonColor"," ","75%"
```

The `changedictionaryname` statement changes the key while leaving the value the same.

```
changedictionaryname dictionary,oldkey,newkey
```

For example, you could change the **Background** key to **BgColor**.

```
changedictionaryname ColorPalete,"Background","BgColor"
```

The value (in this case **white**) will remain the same.

Looking Up a Dictionary Key Given Its Value

In some cases it may be possible to look up a key if all you know is the value. This is the reverse of the normal operation, which is to look up the value given the key. The **getdictionarykey** statement has three parameters:

```
getdictionarykey dictionary,value,key
```

The **dictionary** parameter is of course the name of the field or variable that contains the dictionary. The **value** parameter is the value you are looking for. If this value is not unique (if more than one key/value pair have the same value) then it is unpredictable what key will be returned, so you should only use this statement if you know that the value is unique. The **key** parameter is the name of a field or variable that will be filled with the key name (if any matching value is found, otherwise it will be filled with "").

The example below will find out what key has been set to the value **white**. If this dictionary is set up as it was in the previous examples, the **item** variable will be filled with the value **Background**. (This isn't really a great example, since more than one item could be assigned the color white. If you use this statement, be very careful that you know that there are no duplicate values in the dictionary.)

```
local item  
getdictionarykey ColorPalette,"white",item
```

One other point to keep in mind about this statement — it's a bit on the slow side. It is much slower than the **getdictionaryvalue()** function.

Accessing the Internet

In this millennium no computer is an island. Panorama allows you to tap into the Internet to automatically retrieve data and graphics from the web, submit information to web servers, and to send e-mail based on database information.

Basic Web Access

In this section we'll show how to automatically receive and send data from anywhere on the web.

Fetching Web Pages

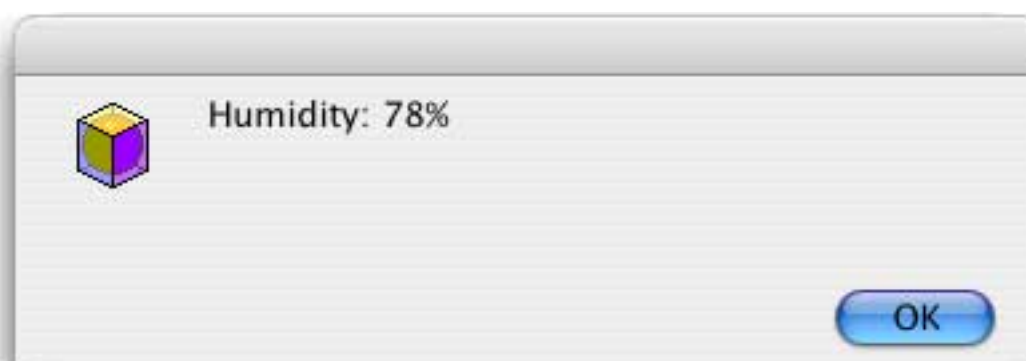
Fetching a web page is simple. The `loadurl` statement has two parameters:

```
loadurl page,url
```

The `page` parameter is the name of the field or variable to store that you want the downloaded page stored in. The `url` parameter is the web address of the page you want to load, for example "<http://www.weather.com>". This example will download the current weather for the 92648 zip code and display the current humidity.

```
local weatherURL,weatherPage,zip
zip="92648"
weatherURL="http://http://www.wunderground.com/cgi-bin/findweather/getForecast?query="
loadurl weatherPage,weatherURL+zip
humidity=tagdata(weatherPage,"Humidity:</td><td valign=top><b>","</b>",1)
message "Humidity: "+humidity
```

When this procedure is triggered the current humidity will be displayed.

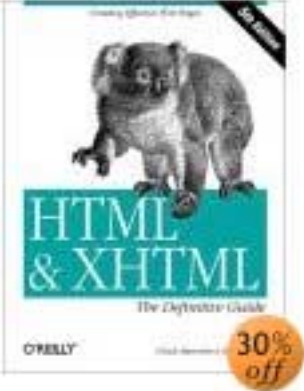



If there is a problem (for example the requested page doesn't exist, or the computer is not currently connected to the internet) the statement will place "" in the page field or variable. This revised example checks to make sure that it got a response from the Weather Underground server.

```
local weatherURL,weatherPage,zip
zip="92648"
weatherURL="http://http://www.wunderground.com/cgi-bin/findweather/getForecast?query="
loadurl weatherPage,weatherURL+zip
if weatherPage<>""
    humidity=tagdata(weatherPage,"Humidity:</td><td valign=top><b>","</b>",1)
    message "Humidity: "+humidity
else
    message "Sorry, weather information is not currently available."
endif
```

Parsing Web Pages

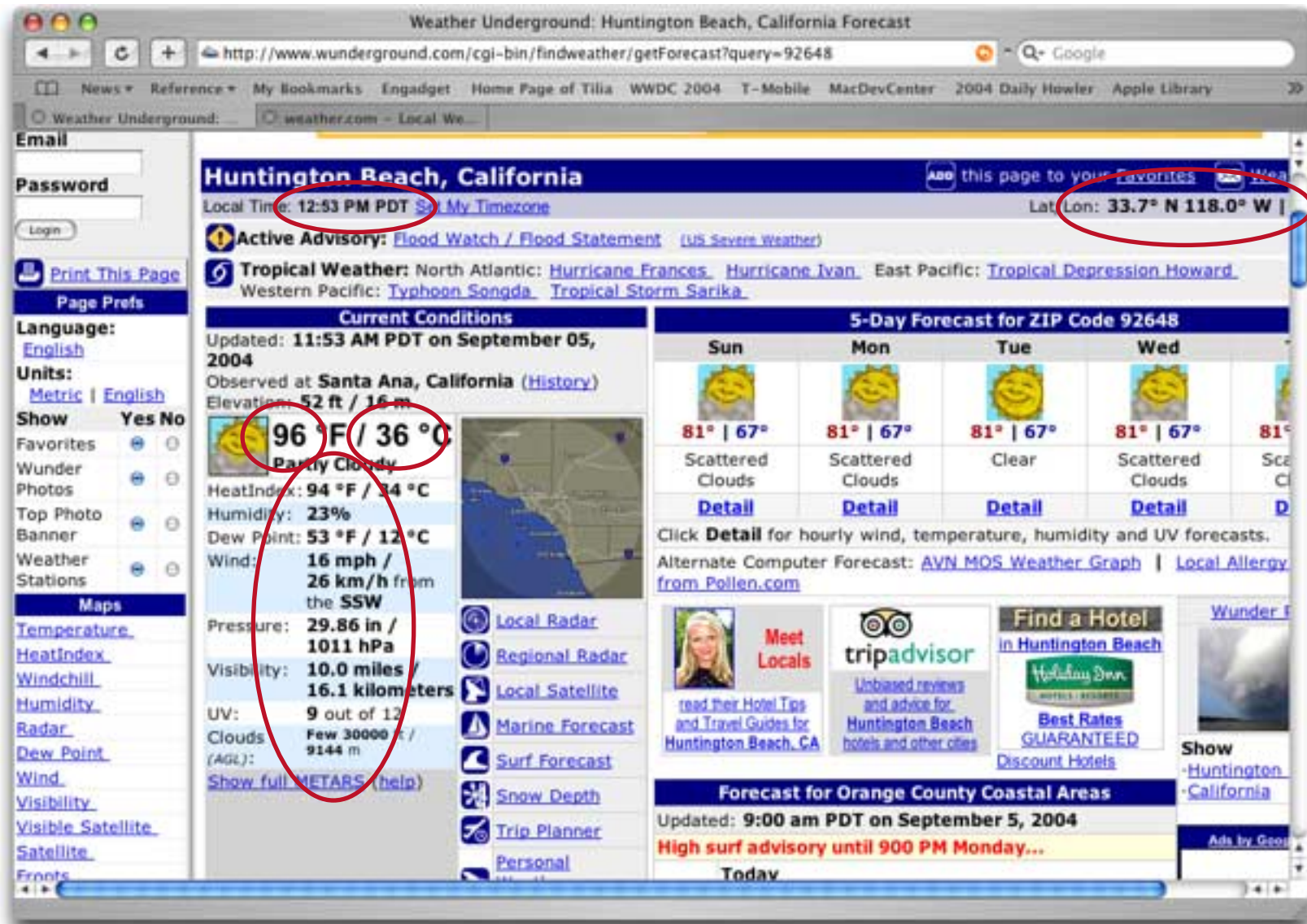
The primary language of the web is HTML (HyperText Markup Language). When you fetch a page from a web server, the page will be constructed with HTML tags like ``, `<a>`, ``, `<input>`, etc. In order to extract the information you need from downloaded web pages you'll need to learn at least some HTML. The details of HTML syntax and vocabulary are beyond the scope of this book, but here are some books that you may find useful.

	<p>HTML & XHTML: The Definitive Guide, Fifth Edition by Chuck Musciano, Bill Kennedy</p> <ul style="list-style-type: none"> • Paperback: 670 pages • Publisher: O'Reilly; 5 edition (August, 2002) • ISBN: 059600382X • Product Dimensions: 9.3 x 7.0 x 1.3 inches <p>HTML: The Definitive Guide is aimed at beginners as well as those who have more practice in Web-page creation. The authors assume at least a basic knowledge of computers, including how to use a word processor or text editor and how to deal with files. They teach you that learning HTML is like learning any other language and that reading a book of rules can only take you so far. Readers begin writing what may be their first Web page just two pages into the book's second chapter. From there on, they provide a wide range of HTML coding to allow readers to learn from good examples. The book includes a handy "cheat sheet" of HTML codes for quick reference.</p>
	<p>Learning Web Design, 2nd Edition by Jennifer Niederst</p> <ul style="list-style-type: none"> • Paperback: 488 pages • Publisher: O'Reilly; 2 edition (June 27, 2003) • ISBN: 0596004842 • Product Dimensions: 9.8 x 8.1 x 1.1 inches <p>In Learning Web Design, author Jennifer Niederst shares the knowledge she's gained from years of web design experience, both as a designer and a teacher. This book starts from the beginning-- defining the Internet, the Web, browsers, and URLs-- so you don't need to have any previous knowledge about how the Web works. After reading this book, you'll have a solid foundation in HTML, graphics, and design principles that you can immediately put to use in creating effective web pages. In the second edition, Jennifer has updated the book to cover style sheets and reflect current web standards. She has also added exercises that help you to learn various techniques and short quizzes that make sure you're up to speed with key concepts. The companion CD-ROM contains material for all the exercises in the book. Unlike other beginner books, Learning Web Design leaves no holes in your education. It gives you everything you need to create basic web sites and will prepare you for more advanced web work.</p>

When you fetch a web page with Panorama it's usually because you want to retrieve one or more specific pieces of information within that page. Often the specific information you want is buried like gold nuggets in raw ore. Instead of using a pan or sluice to extract the nuggets, you'll use statements and functions like `search()`, `tagdata()`, `gettaglocation` and `htmltabletoarray`. Extracting information from web pages is called **web scraping**, and you can find various books and articles on the subject in your local bookstore, in programming magazines, and on the web.

Web scraping is kind of like detective work. The trick is to see through the clutter of tags and recognize the patterns that can reliably point to the nuggets of information you are interested in. The detective has tools that help solve the case — DNA, fingerprints, chemical analysis, phone records etc. But it takes imagination and skill to use these tools to connect the dots. Similarly, we can teach you how to use the tools Panorama makes available for analyzing and parsing web pages. But every page is unique and you'll need to develop your own sixth sense for teasing out the patterns in the tags.

In the previous section an example fetched the current weather for the 92648 zip code. In your web browser, this page might look something like this:



As you can see, there is all kinds of interesting data that could be extracted from this page — the current time, the latitude and longitude of this location, the current temperature (in Fahrenheit and Celsius), the humidity, wind, pressure, and many other items. The page layout is designed to make it easy to pick this information out when displayed in a browser. In the actual HTML source, however, it is much more difficult to locate this information. Here we've used the browser's **View Source** command to examine the underlying HTML code for this page:

```

</tr><tr>
<td valign=top width="100%">
  <table class="smalltable" cellspacing=1 cellpadding=1 border=0 width="100%">
  <tr valign=center bgcolor=#ffffff>
  <td width=42>
  
  </td>
  <td>
  <font size=+2 face=arial><b>
  <nobr><b>97</b></b>&nbsp;&#176;F</nobr>
  /
  <nobr><b>36</b></b>&nbsp;&#176;C</nobr>
  </b></font><br>
  <font size=-1><b>Partly Cloudy</b></font>
  </td>
  </tr>
  </table>
<table cellpadding=1 cellspacing=0 border=0 class=smalltable width="100%">
  <tr bgcolor=#FFFFFF><td valign=top>
  Humidity:</td><td valign=top><b>10%</b>
  </td></tr>
  <tr bgcolor=#ddeeff><td valign=top nowrap>
  Dew Point:</td><td valign=top><b>
  <nobr><b>32</b></b>&nbsp;&#176;F</nobr>
  /
  <nobr><b>0</b></b>&nbsp;&#176;C</nobr>
  </b>
  </td></tr>
  <tr bgcolor=#FFFFFF><td valign=top>
  Wind:</td><td valign=top>
  <b>
  <nobr><b>10</b></b>&nbsp;&#176;mph</nobr>
  /
  <nobr><b>17</b></b>&nbsp;&#176;km/h</nobr>
  </b> from the <b>SW</b>
  </td></tr>
  <tr bgcolor=#ddeeff><td valign=top>
  Pressure:</td><td valign=top><b>
  <nobr><b>29.86</b></b>&nbsp;&#176;in
  /
  <nobr><b>1011</b></b>&nbsp;&#176;hPa
  </td></tr>
  </table>

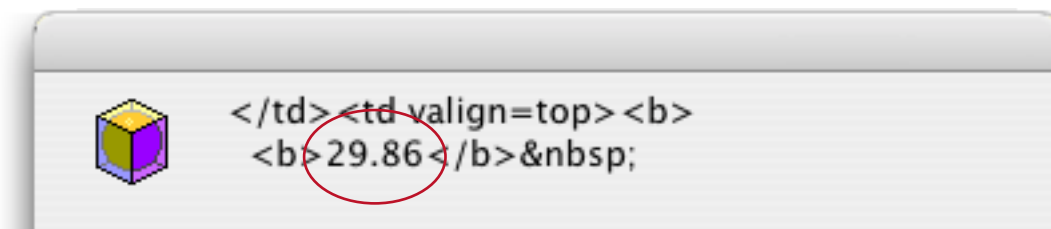
```

You're first task is to visually locate the information you are looking for. In some cases this is fairly easy and you can use your browser or text editor's **Search** command to directly locate the information, for example the Humidity, Dew Point, Wind and Pressure in the text above. In other cases, however, it can be more tricky. There is no clearly identifying text near the temperature readings. In cases like this you'll have to find the information by looking at the original page for other items near the item you are looking for. For example, looking at the original page we can see that the temperature readings are just above the humidity measurement. The humidity is easily found, so looking above we quickly find the temperature.

Once you have managed to find the information visually the real trick begins — devising a means to automatically locate this information with a program. Look around for something unique that can be easily found. For example, the text **Pressure:** only occurs in one place in this web page, so it can be used with the `tagdata()` function to help locate the barometric pressure measurement.

```
message tagdata(weatherPage,"Pressure:","in",1)
```

As you can see, this formula has removed almost all of the text except for a few tags around the pressure.



There are several methods that could be used to isolate just the pressure itself. Noticing that none of the tags that are left contain any numbers or periods, we can use the `stripchar()` function to extract just the pressure.

```
stripchar(tagdata(weatherPage,"Pressure:","in",1),"09..")
```

There is usually more than one method that could be used to extract a particular text item. For example, here is another method that you could use:

```
tagdata(weatherPage,"Pressure:</td><td valign=top><b>"+lf()+" <b>","</b>&nbsp;in",1)
```

Usually the simplest formula is the best, but not always. The goal is to create a method that is as reliable as possible, a method that will hopefully continue to work even if there are minor changes made to the page design. Of course it's always possible that the site owners will completely redesign the page or even their entire site at some point in the future. At that time you'll also have to redesign your web scraping method. Keeping your design modular will help minimize the work necessary when a redesign is necessary.

Panorama has several dozen statements and functions that can help you scrape information from web pages. Use the **Programming Reference** wizard to study these tools before you begin your first web scraping project.

Statement/Function	Description
<code>gettaglocation</code>	This statement finds the position of the first matching tag within some text. For example, you could find the position of the first <code><table></code> tag, or of the word <code>Pressure</code> .
<code>gettaglocations</code>	This statement finds the positions of all matching tags within some text. For example, you could find the position of every <code></code> tag (not just the first one).
<code>htmldecode()</code>	This function takes HTML text and converts any special characters in the text into standard ASCII. For example <code>&copy;</code> is converted to © and <code>&amp;</code> is converted to & .
<code>htmlextractlinks</code>	This statement extracts the links from an HTML page. The result is a carriage return separated list of the links (URL's) from this page to other pages.
<code>htmlformitemnames</code>	This statement returns a list of form elements (<code><INPUT</code> tags, etc.). The result is a carriage return separated list of these elements. This statement is very handy for extracting any data that may be included in an HTML form.
<code>htmltablecell()</code>	This function extracts the data from a cell in an HTML table. Any HTML tags in the cell are removed, leaving only the actual text.
<code>htmltablecellexists()</code>	This function checks to see whether a cell in an HTML table exists or not. The result will be true if the cell exists, or false if it doesn't.

Statement/Function	Description
htmltablecellraw(This function extracts the data from a cell in an HTML table. Unlike the htmltablecell(function, any HTML tags in the cell are retained.
htmltableheight(This function calculates the height (number of rows) in an HTML table. It assumes that the table is a regular matrix (no rowspan tags).
htmltablerowraw(This function extracts the data from a row in an HTML table. Any HTML tags in the row are retained.
htmltabletoarray	This statement converts an HTML table into a text array. The HTML table must be a simple array with no nested table inside it. You can include all of the columns in the final array, or just selected elements.
htmltablewidth(This function calculates the height (number of rows) in an HTML table. It assumes that the table is a regular matrix (no colspan tags) and that all of the rows have the same number of columns as the top row in the table.
onespace(This function removes any extra spaces between words, so that there is exactly one and only one space between each word.
onewhitespace(This function removes any extra whitespace between words, making sure that there is one and only one space between each word. Other whitespace characters (carriage returns, tabs) are converted to spaces and removed if there is more than one between words.
stripchar(This function removes characters you don't want from a text item. You specify exactly what kinds of characters you want and don't want included in the final output.
tagarray(This function builds an array containing the body of all the specified tags (usually HTML tags) in the text. Each element in the array is separated from the next with the separator character (usually ¶ or ,).
tagcount(This function counts the number of times a specified tag (usually an HTML tag) appears in the text.
tagdata(This function extracts the body of the specified tag (usually an HTML tag) in the text.
tagend(This function returns the ending position of the specified tag (usually an HTML tag) in the text.
tagnumber(This function checks to see if a specified position is inside of a tag (usually an HTML tag).
tagparameter(This function extracts the value of a tag parameter embedded in some text, where the tag parameter takes the form name=value .
tagparameterarray(This function extracts the value of multiple tag parameter embedded in some text, where each tag parameter takes the form name=value .
tagstart(This function returns the starting position of the specified tag (usually an HTML tag) in the text.
tagstrip(This function removes tags (usually an HTML tag) from within a piece of text.
textafter(This function extracts the text after the tag. The tag may be one or more characters long. If the tag doesn't occur in the text then the entire original string is returned.
textbefore(This function extracts the text before the tag. The tag may be one or more characters long. If the tag doesn't occur in the text then the entire original string is returned.
urldecode(This function takes standard ASCII text and converts into a format guaranteed to be legal in an internet URL (Universal Resource Locator). For example the url my%20web%20page is converted to my web page .

In addition to these statements and function that are specifically designed for HTML parsing you may also need Panorama's general text handling tools. See "[Text Formulas](#)" on page 67.

Fetching Images

Fetching an image is simple. The `saveurl` statement has two parameters:

```
saveurl path,url
```

The `path` parameter is the path and filename of the file that you want the downloaded image stored in. The `url` parameter is the web address of the image you want to load, for example `"http://icons.wunderground.com/graphics/conds/clear.GIF"`. (This statement isn't limited to images, you can also save a regular HTML web page directly into a file.) This example will download the icon for "sunny and clear" from the weather underground web site.

```
saveurl "SunnyAndClear.gif", "http://icons.wunderground.com/graphics/conds/clear.GIF"
```

Since the path includes only a filename (no disk or folders) the file will be stored in the same folder as the database. If you have the Enhanced Image Pack (see "[Displaying Non PICT Images \(Enhanced Image Pack\)](#)" on page 775 of the *Panorama Handbook*) you can display this GIF image in a form.



If you want to save the image in a subfolder of the current database folder then the path must begin with a colon. This example will save the image in the Icons folder which is in the same folder as the current database.

```
saveurl ":Icons:SunnyAndClear.gif",
"http://icons.wunderground.com/graphics/conds/clear.GIF"
```

You can also specify a complete path to store the image anywhere on your hard drive.

```
saveurl "My Disk:Documents:Web Icons:SunnyAndClear.gif",
"http://icons.wunderground.com/graphics/conds/clear.GIF"
```

No matter where the path points to, the folder must already exist. If it doesn't you can use the `makenewfolder` statement to create it.

Relative URLs

The `loadurl` and `saveurl` statements normally require a complete URL, like this:

```
loadurl thePage, "http://www.someserver.com/somefolder/somepage.html"
```

However, once you have loaded or saved a URL you can then access other url's relative to that URL. For example, suppose that there is an image in the same folder as `somepage.html`. You could fetch this image with an absolute URL like this:

```
saveurl "someimage.jpg", "http://www.someserver.com/somefolder/someimage.jpg"
```

However, if you have just loaded the `somepage.html` page with `loadurl`, you can then save the image with a relative URL like this:

```
saveurl "someimage.jpg", "someimage.jpg"
```

You don't need to specify the complete URL because Panorama assumes that the missing part is the same as the last URL. You can also use the common `../` prefix notation to move up the server's directory structure. Again assuming that you have just loaded the `somepage.html` page, then these two URL's are the same.

```
"http://www.someserver.com/differentfolder/bigimage.jpg"
"../differentfolder/bigimage.jpg"
```

Remember, relative URL's are relative to the last URL saved or loaded.

Submitting Forms

Many applications call for submitting information to a web site. This is called a "form." An HTML form contains one or more fields that are submitted to the web server. When using a web browser, you fill out a form by typing and/or by selecting from pop-up's, checkboxes and radio buttons. When you have entered all of the information you press the Submit button.

When using Panorama you can submit information to a web site using the `posturl` statement.

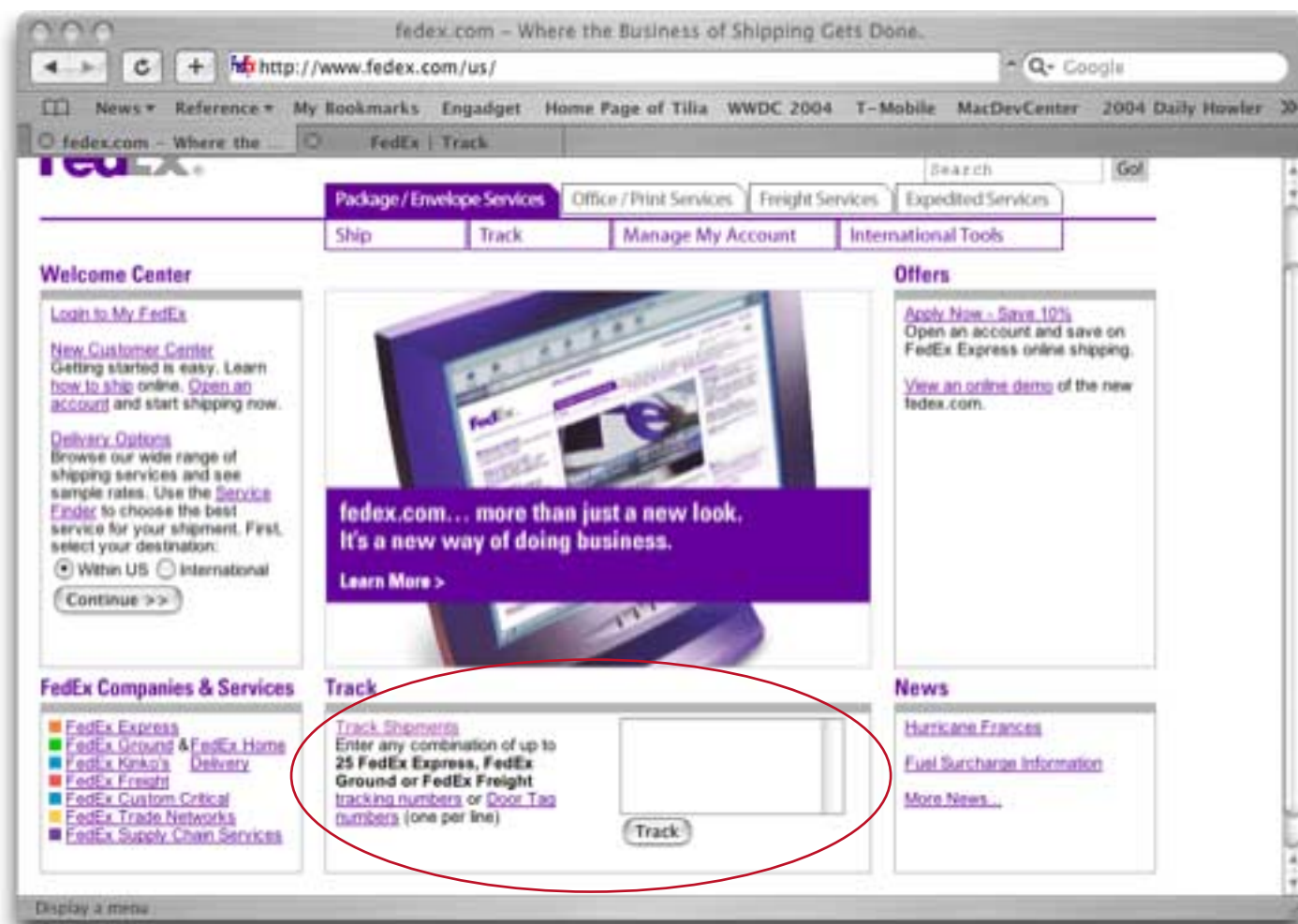
```
loadurl page,url,field1,value1,field2,value2,field3,value3 ...
```

The `page` parameter is the name of the field or variable to store that you want the downloaded page stored in. This is the server's response to the data you have submitted. The `url` parameter is the web address of the page you want to load, for example "<http://www.fedex.com/cgi-bin/tracking>". The field parameters are the names of the data items being submitted to the server, while the value parameters supply the actual data for each corresponding field. The statement can have as many field/value pairs as you need, but they must come in pairs. This example will look up shipment information for a FedEx tracking number. The tracking number must be in a field or variable named `trackingNumber`.

```
posturl webPage,"http://www.fedex.com/us/tracking/",
  "tracknumbers",trackingNumber,
  "action","track",
  "language","english",
  "cntry_code","us",
  "mps","y",
  "ascend_header","1"
```

In this case we are submitting six fields to the FedEx server. If we don't submit the exact fields the FedEx server is expecting to the right URL, the FedEx server will not respond. How can we find out what fields we need to submit?

To find out, first use your browser to go to the web server you want to access, and find the form you want to submit using Panorama.



Now use View Source to look at the actual HTML source code and find the HTML code for the form you want to submit.

```

Source of http://www.fedex.com/us/

<td class="small">
  <form id="tracking" method="post" action="/us/tracking/">
    <textarea style="font-family: monospace; font-size: 11px;
width: 150px; height:65px;" name="tracknumbers" cols="18" rows="3"></textarea>
    <input type="hidden" name="action" value="track">
    <input type="hidden" name="language" value="english">
    <input type="hidden" name="cntry_code" value="us">
    <input type="hidden" name="mps" value="y">
    <input type="hidden" name="ascend_header" value="1">
    <input type="submit" name="imageField" value="Track"
class="buttonpurple">
  </form>
</td>

```

We've underlined the important parts of this form as far as Panorama is concerned. The `action=/us/tracking` parameter of the `<form>` tag is added to the base URL for this page to form the URL for submitting the form

<http://www.fedex.com/us/tracking/>

In this form the `<textarea>` tag contains the `name=tracknumbers` of the field that is used for submitting the tracking numbers. This becomes the first field parameter to our `posturl` statement (see above). All of the other five `<input>` tags are hidden and have fixed tags, so these make up the additional `posturl` parameters.

In this case the FedEx web authors created a nice compact form that is easy to decipher. Often forms will not be this compact - you'll have to wade through tons of additional tags and whitespace. Just let the tags guide you — start by finding the opening and closing `<form>` tags (and remember that some pages have more than one form on the page, so you have to make sure you have the right form). The opening `<form>` tag will contain the action parameter, which will have a relative or absolute URL. Then look for `<input>`, `<textarea>` and `<select>` tags to find out the data that needs to be submitted.

Once you've got the `posturl` statement working you are still not quite done. The server will return an HTML page that contains the information you want to retrieve (if any - sometimes you may just want to submit information, for example to add information to an on-line database). If you need to retrieve information from this response you'll have to use the web scraping techniques discussed earlier in this chapter. (However, for FedEx tracking numbers we've already done this work for you as you'll see later in this chapter.)

Cookies

A “cookie” is a small piece of information sent by a web server to store on a web client so it can later be read back from that browser. This is useful for having the browser remember some specific information, for example user id's, shopping baskets, and site preferences. By default Panorama ignores cookies, but some web pages cannot be accessed unless the web client handles cookies. If you are accessing a site that requires cookies you can turn them on with the `cookies` statement. (Note: Currently only the OS X version of Panorama supports this statement.) For most applications you can simply place the `cookies` statement with no parameters near the top of your procedure, before any `loadurl` or `posturl` statements that would require cookies to be saved.

```
cookies
loadurl ...
posturl ...
```

The `cookies` statement also has three optional parameters that give you more control over how cookies are processed and stored.

```
cookies folder,file,options
```

The `cookies` statement keeps the information it gets from the web server in a file on your hard disk. The `folder` and `file` parameters specify where this file should be stored and what name should be used. If the `folder` parameter is empty, Panorama will keep the cookie file in the same folder as the current database. If the `file` parameter is empty Panorama will automatically assign a name for you.

The `options` parameter allows you to specify one or more options. Each option is specified by a keyword from the table below.

Keyword	Description
global	By default cookies are only enabled for one database at a time. If you enable cookies for more than one database, each will keep a separate cookie file. However, if you include the <code>global</code> keyword in the options parameter Panorama will create a global cookie file that will be used no matter what database is currently active. If you don't specify the file and folder options this file will be called <code>Panorama.cookie</code> and will be stored in the <code>Extensions:Work Files</code> folder.
suspend	The cookies statement normally turns cookies on, but if this keyword is included cookies will be suspended. After this option is used any cookies information sent from the server will be ignored. You can re-enable cookies later, any cookie information that was already saved when cookies were suspended will be retained.
reset	The <code>reset</code> option completely erases any cookie information that has been saved. If you want to erase this information and turn cookies off, use this option in combination with the <code>suspend</code> keyword. If you want to erase global cookie information you must include the <code>global</code> keyword as well.

These keywords can be combined, for example “[global suspend reset](#)” to stop and erase any global cookies that have been stored.

Accessing Web Content

The previous section described how to perform general access to any web server. Panorama also includes pre-built tools designed for accessing specific types of information, for example maps, addresses, shipping information, etc. We will be adding new tools for accessing web content on a regular basis, so check our web site for updates.

Generating Map Images

The [savewebmap](#) statement makes it easy to download a map for any US address. If you have purchased the Enhanced Image Pack you can display this image on a form. This statement has eight parameters.

```
savewebmap path, zoom, street, city, state, zip, country, mapurl, mapheight, mapwidth
```

The [path](#) parameter specifies the name and location of the image file to save. For example, if you wanted to save the file in the folder that contains the current database this parameter only needs the file name:

```
savewebmap "Acme Widgets Map.jpg", ...
```

If you want to store the image file in a subfolder of the current database folder, start the path with a colon (:).

```
savewebmap ":Maps:Acme Widgets.jpg", ...
```

You can also specify a full path, including the disk name, to place the saved image anywhere on your hard drive.

The image doesn't have to be a .jpg file. You can also specify a file with the .png or .gif extensions.

The [zoom](#) parameter is the map zoom level. There are two ways to specify the zoom level. You can use a nine step scale from **1** (entire country) to **9** (a few blocks). If you want more control you can also directly specify the denominator of a scaling ratio, for example **500000** for a 1:500,000 scale (approximately city or county level) or **2000** for a 1:2000 scale (zoomed in to a few blocks). Using a ratio allows you to specify any zoom level you want. The nine standard ratios are:

Level	Ratio
9	1:3900
8	1:13,600
7	1:45,500
6	1:116,000
5	1:516,000
4	1:1,160,000
3	1:4,700,000
2	1:12,600,000
1	1:29,000,000

If no zoom level is specified the default is **8** (ratio of 1:13,600).

The [street](#), [city](#), [state](#), [zip](#) and [country](#) parameters specify the address you want to map. Do not include the suite or apartment number in the [street](#) address. The [country](#) parameter is currently ignored, this statement only works for United States. Future versions may be more global.

The [mapurl](#) parameter must contain the name of a field or variable. When the statement is complete this field or variable will contain the url of the map that was saved. If the address was not a valid address the field or variable will be empty ("").

The `mapheight` and `mapwidth` parameters specify the dimensions of the generated map (in pixels). These parameters are optional — if omitted the default height and width supplied by the web site generating the map (currently Yahoo) will be used.

This example creates a .jpg file that contains a map showing the location of Disneyland.

```
local mapurl
savewebmap "Disneyland Map.jpg",7,
    "1313 S. Harbor Blvd.", "Anaheim", "CA", "92803", "", mapurl,400,600
```

Here is the result of this program (displayed in Apple's **Preview** program):

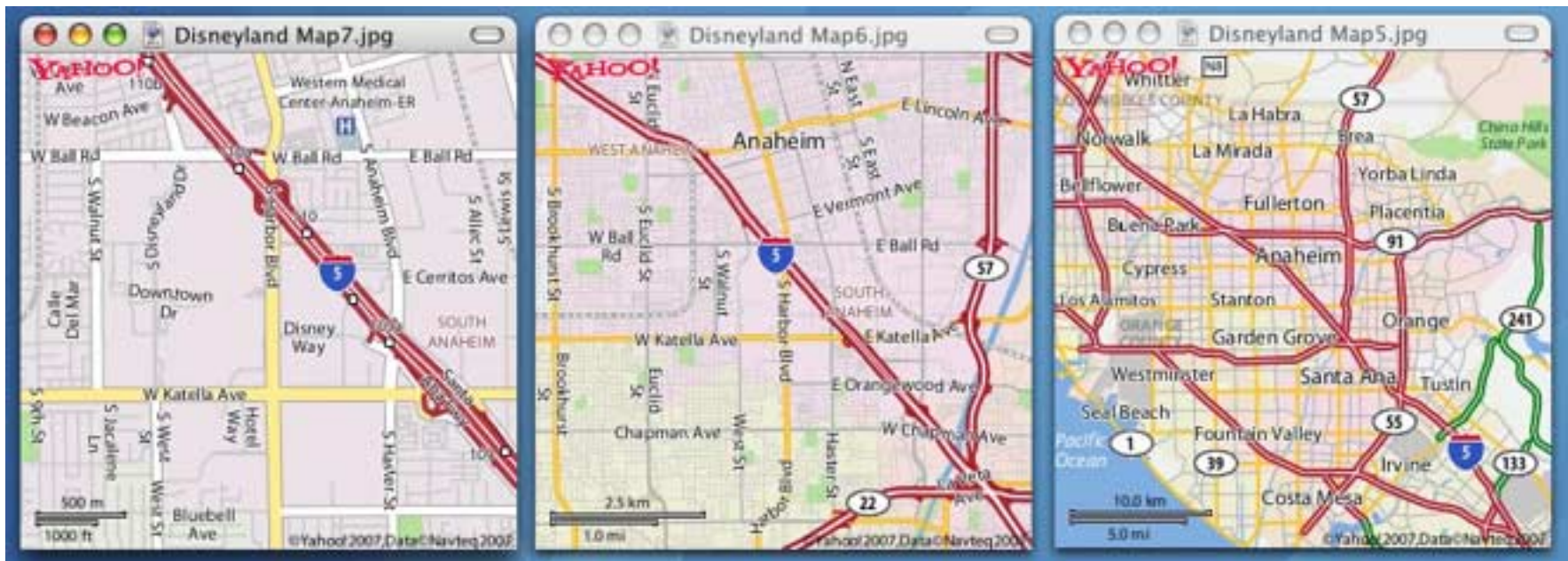


Note: The `savewebmap` statement works by requesting maps from a web server. The current version of this statement uses Yahoo! Maps (earlier versions used MapQuest). In the future the operation of this statement may change based on changes to the underlying Yahoo! Maps server.

Generating the Same Map at Different Zoom Levels (Scales) or Sizes. A shortcut makes it faster to regenerate a map with a different zoom level or size than to generate an all-new map. To use this shortcut take the result of the `mapurl` parameter and feed it back into the street parameter the next time you generate the map. Here is an example that generates three different maps for Disneyland at different zoom levels.

```
local mapurl,mapzoom
mapzoom=7
savewebmap "Disneyland Map"+str(mapzoom)+".jpg",
    mapzoom,"1313 S. Harbor Blvd.", "Anaheim", "CA", "92803", "", mapurl,240,300
loop
    mapzoom=mapzoom-1
    savewebmap "Disneyland Map"+str(mapzoom)+".jpg",mapurl,"","","",mapurl,240,300
while mapzoom>5
```

Here are the three maps generated by this program.



Adding an Interactive Map Interface to a Database

You can use the `savewebmap` statement with Super Flash Art object on a form to create a custom interactive map. If you are in a hurry and are willing to have the map appear in a separate window, however, Panorama can do virtually all of the work for you. All you need to do is add one line of code to your program using the `openmapwindow` statement. This statement has 8 parameters:

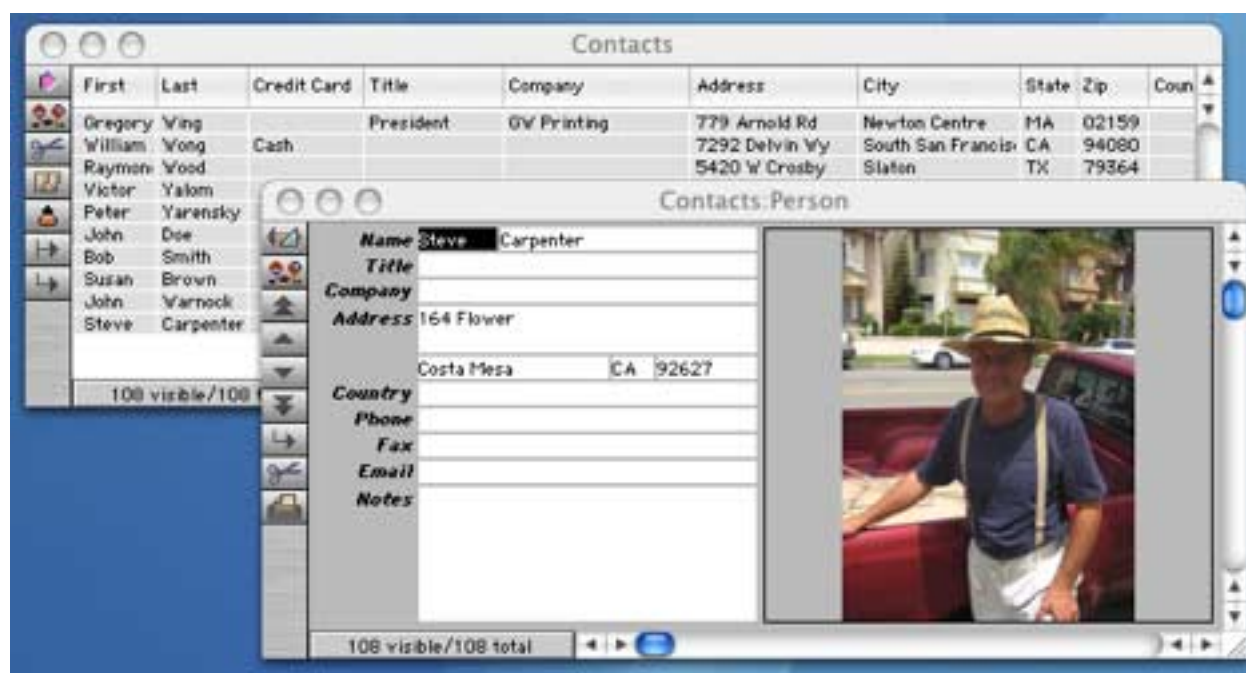
```
openmapwindow street,city,state,postalcode,country,mapheight,mapwidth,options
```

The `street`, `city`, `state`, `zip` and `country` parameters specify the address you want to map. Do not include the suite or apartment number in the `street` address. The `country` parameter is currently ignored, this statement only works for United States. Future versions may be more global.

The `mapheight` and `mapwidth` parameters specify the initial dimensions of the map window (in pixels). These parameters are optional — if omitted the default height is 500 pixels and the default width is 620 pixels.

The `options` parameter is for additional options. Currently this parameter is not used and is ignored.

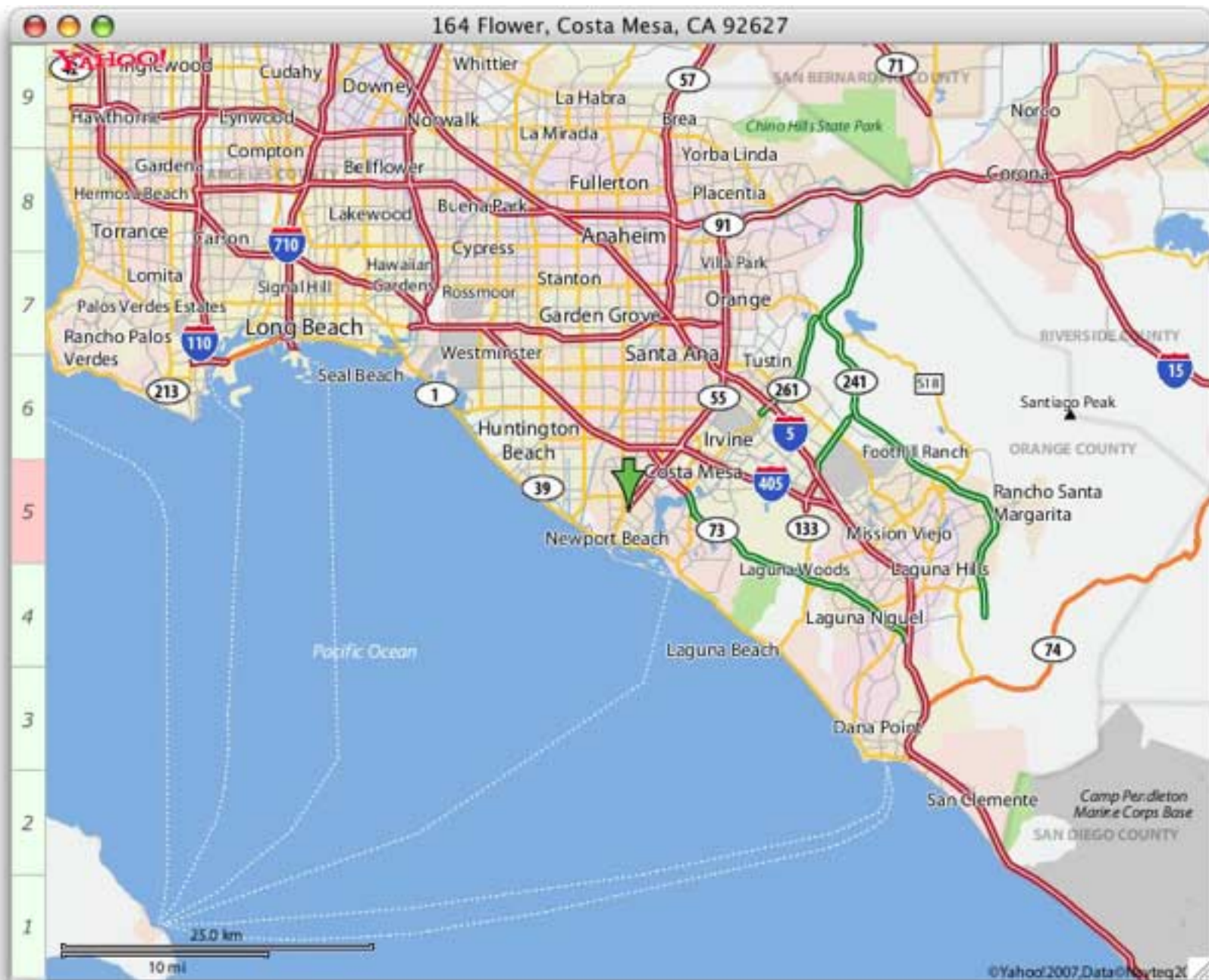
Lets look at how this statement could be used in a contact database. Start with any database that contains US address information, like this:



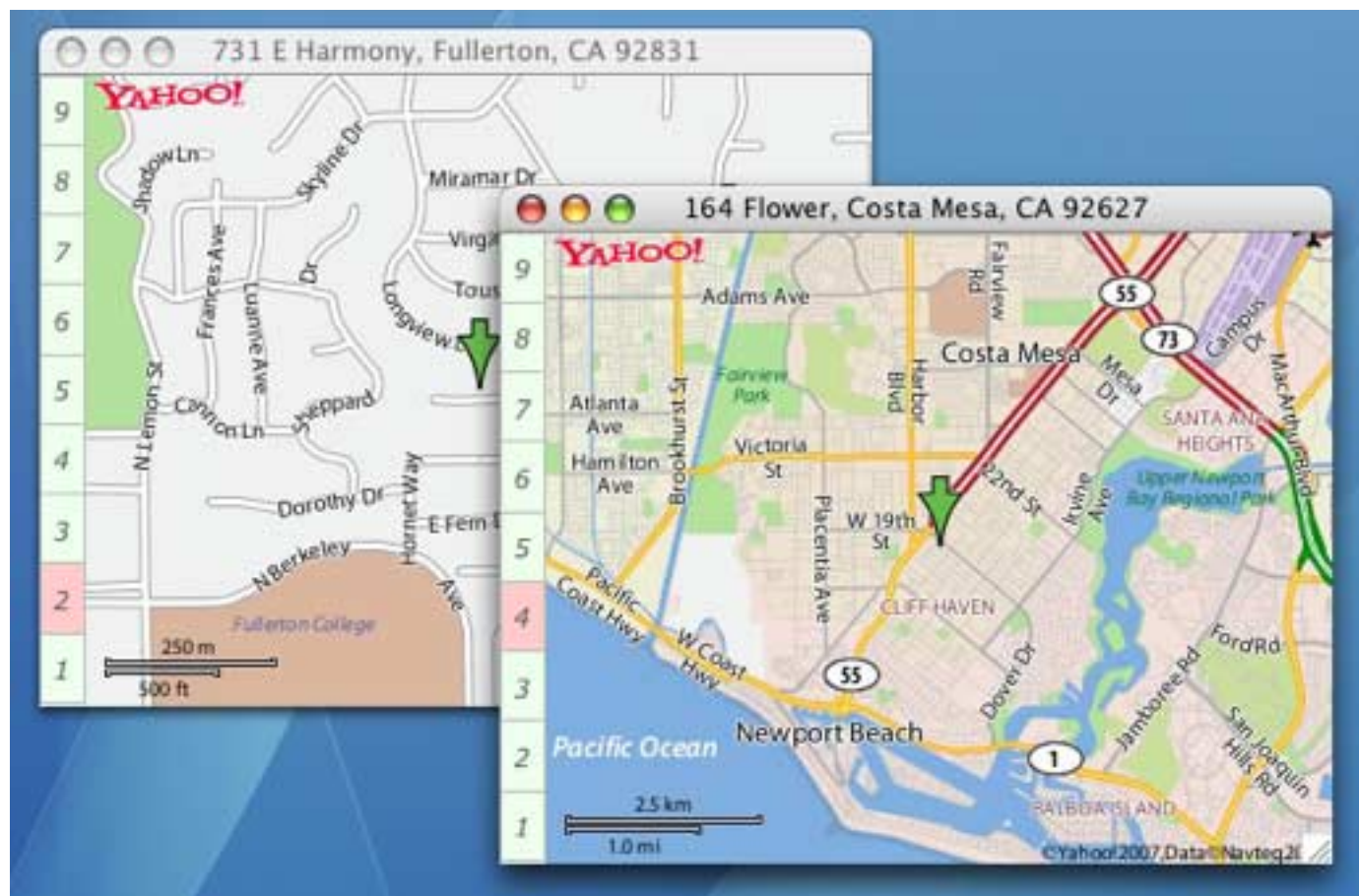
You can zoom in and out by clicking on the numbers on the left edge of the map.



You can also resize the window. Making the map larger will increase the area displayed, and in some cases increase the amount of detail displayed as well.



You can open additional maps without closing the first map (up to Panorama's 64 window limit).



To open additional maps simply navigate within the database to the address you want to see, then run the procedure that contains the `OpenMapWindow` statement again. Each map window can be independently zoomed and resized. When you are done with a map window simply close it.

General Zip Code Information

The `zipinfo` statement queries the web to get general information about a zip code: city, state, area code, etc.

```
zipinfo zipcode,info,channel
```

The `zipcode` parameter must contain a 5 digit zip code. This must be a text value, not a number (e.g. "95001" not 95001).

The `info` parameter must be the name of a field or variable. When the statement is finished this field or variable will contain a data dictionary that contains the information loaded from the web (see "[Data Dictionaries](#)" on page 602). Depending on the channel you have selected (see below) this data dictionary will contain one or more of the items listed below. You can retrieve individual items from this dictionary with the `getdictionaryvalue()` function.

Item	Description	Channel	
		USPS	ZipInfo.com
CITY	Primary name of the city associated with this zip code.	√	√
STATE	Two letter abbreviation of the state this zip code is in.	√	√
COUNTY	The name of the county this zip code is in.	√	√
FIPS	FIPS code.		√

Item	Description	Channel	
		USPS	ZipInfo.com
AREACODE	Name or the primary area code associated with this zip code.		√
TIMEZONE	Time zone associated with this zip code.		√
DAYLIGHTSAVINGS	Yes/No		√
LATITUDE	Geographic co-ordinates of center of zip code.		√
LONGITUDE	Geographic co-ordinates of center of zip code.		√
MSA	Metropolitan statistical area		√

This statement uses a channel to retrieve the zip code information from the web. You can set the default channel using the **Channels** wizard. You can use the `channel` parameter to override the default. At the time of this writing the options for this parameter are "USPS" and "ZipInfo.com", but additional options may be available in the future. Check the Channels wizard for the latest information.

This example displays the city and state associated with a zip code.

```
local z5,zinfo
z5=""
gettext "Zip Code:",z5
zipinfo z5,zinfo
message z5+": "+getdictionaryvalue(zinfo,"CITY")+", "+getdictionaryvalue(zinfo,"STATE")
```

Street Address Information

Given a US street address, the `zipinfoplus` statement uses the US Post Office web site to find out the zip+4, carrier route, and other information.

```
zipinfoplus address1,address2,city,state,zip5,info
```

The `address1` parameter is the first line of the address, `address2` is the second line (optional). The `city` parameter is the name of the city, `state` is the two letter state abbreviation. The `zip5` parameter is the five digit zip code, but you can leave this blank.

The `info` parameter must be the name of a field or variable. When the statement is finished this field or variable will contain a data dictionary that contains the information loaded from the web (see "[Data Dictionaries](#)" on page 602). This data dictionary will contain items listed below. You can retrieve individual items from this dictionary with the `getdictionaryvalue()` function.

Item	Description
ADDRESS	Corrected street address.
CITY	Primary name of the city associated with this address.
STATE	Two letter abbreviation of the state this zip code is in.
COUNTY	The name of the county this zip code is in.
ZIP9	Nine digit zip code.
CARRIERROUTE	Postal carrier route.

This example checks an address in the database to see if it is correct. If it is completely invalid, an error message is displayed. If it is not formatted correctly by USPS rules, the database is corrected.

```
local zinfo,correctAddress
zipinfoplus Address," ",City,State,Zip,zinfo
correctAddress=getdictionaryvalue(zinfo,"ADDRESS")
if correctAddress=""
    message "Address is not valid!"
    rtn
endif
if correctAddress=Address rtn endif
Address=correctAddress
```

White Pages

The `querywhitepages` statement queries the web to look up white page information.

```
querywhitepages first,last,city,state,zip,url,listings,info,channel
```

The `first` parameter is the first name of the person you are looking for. You can leave this blank if you don't know the first name, or use a partial first name (for example the first initial). The more detail you can supply, the more likely it is that you will be able to find the person you are looking for.

The `last` parameter is the last name of the person you are looking for. You cannot leave this blank, but you can use a partial name. Again, however, the more you supply the more likely you'll get the information you're looking for.

The `city` is the city you are looking for. You can leave this blank if you don't know, or if you are supplying the zip code.

The `state` is the two letter state abbreviation. You can leave this blank, but unless the name you are looking for is extremely unusual this is generally not a good idea (unless you supply a zip code).

The `zip` code can be supplied instead of the city and state.

The `url` parameter is normally set to "". If it is not blank, it must be a URL received from a previous invocation of the `querywhitepages` statement. This allows you to download additional listings that match your search criteria, and will be described in more detail below.

The `listings` parameter must be the name of a field or variable. When the statement is complete, this field or variable will contain a carriage return separated list of names, addresses and phone numbers. Each line contains information for a single person. Within in each line are seven tab separated fields:

First	Last	Address	City	State	Zip	Phone
-------	------	---------	------	-------	-----	-------

We've found that sometimes entries earlier in the list contain more information than entries farther down. To get the most detail about a specific person, narrow the search as much as possible.

The `info` parameter must be the name of a field or variable. When the statement is complete, this field or variable will contain a data dictionary with the following items, which you can retrieve with the `getdictionaryvalue()` function:

Item	Description
FOUND	This is the total number of people found. This number may be more than the actual number in the list that is returned, this list normally contains no more than 10 lines no matter how many people were found. If this number is -1 then the total number of people found is a very large, but unknown number.
INDEX	This is the index number of the first person in the list of returned names. This value is always 1 unless you have used the <code>url</code> parameter to request additional names (beyond the first ten).
MOREPEOPLEURL	If more than ten people are found, only the first ten are returned in the list. In that case, this value will contain a URL that can be used to request additional names that match this search. You can pass this URL back to this statement using the <code>url</code> parameter (in that case, leave the <code>first</code> , <code>last</code> , <code>city</code> , <code>state</code> and <code>zip</code> parameters blank). Each time you request additional names you can check this data dictionary entry to see if more names are available, and if so, you can request them.

This statement uses a channel to retrieve the white page listings from the web. You can set the default channel using the **Channels** wizard. You can use the `channel` parameter to override the default. At the time of this writing the only option for this parameter is "Switchboard", but additional options may be available in the future. Check the **Channels** wizard for the latest information.

To see examples of how the `querywhitepages` statement can be used see the **White Pages** wizard that comes with Panorama.

FedEx Shipment Tracking

The `fedextracking` statement queries the FedEx web site to determine the status of a shipment.

```
fedextracking trackingnumber,shipinfo
```

The `trackingnumber` parameter must contain a FedEx tracking number.

The `shipinfo` parameter must be the name of a field or variable. When the statement is complete, this field or variable will contain a data dictionary with some of the following items (which items are available depends on the current package status), which you can retrieve with the `getdictionaryvalue()` function:

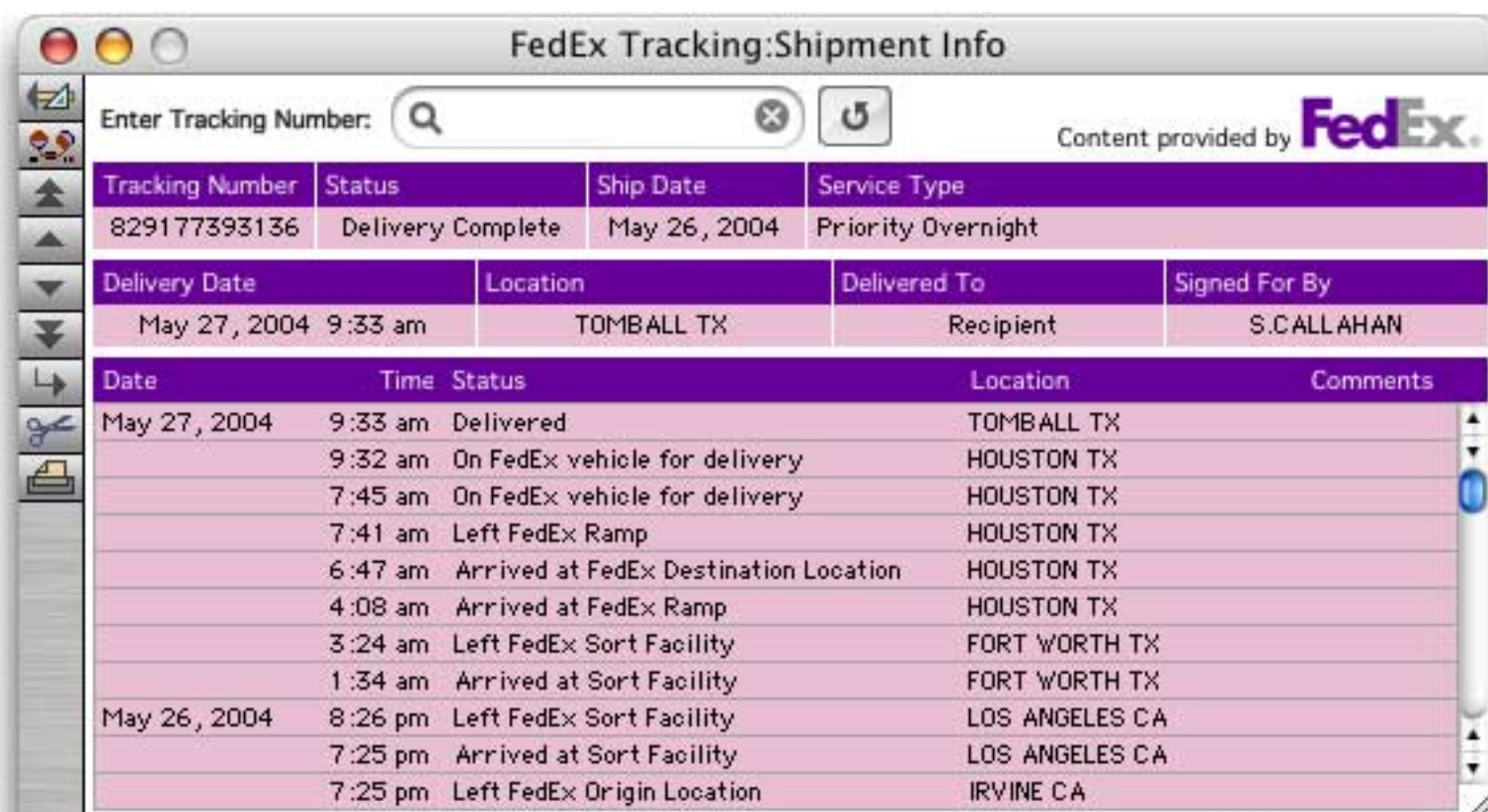
Item	Description
TRACKING NUMBER	This is the tracking number, a duplicate of the first parameter
STATUS	Status of this shipment. Possible values are "Invalid", "Not Shipped", "In Transit" and "Delivery Complete".
SHIP DATE	This is the date the package was picked up by FedEx.
ESTIMATED DELIVERY DATE	If the package is in transit, this is the estimated delivery date and time.

Item	Description
DELIVERY LOCATION	The package destination (city and state).
DELIVERED TO	If the package has been delivered, the location or person it was delivered to (for example "Recipient", "Front Desk" or "Receiving Dock").
SERVICE TYPE	"Priority Overnight", etc.
SIGNED FOR BY	If the package has been delivered, the name of the person that signed for it.
DELIVERY DATE/TIME	If the package has been delivered, the date and time that occurred.
SHIPMENT HISTORY	This is a carriage return separated array showing the progress of the shipment. Each line contains 5 columns separated by tabs: Date, Time, Action/Status, Location and Comments.

This example displays the status of a FedEx shipment.

```
local tracknum,trackinfo
tracknum=""
gettext "Tracking number:",tracknum
fedextracking tracknum,trackinfo
message "Shipment status: "+getdictionaryvalue(trackinfo,"STATUS")
```

For more a more detailed example of this statement see the **FedEx Tracking wizard**.



Controlling Web and E-Mail Clients

In addition to directly accessing web pages (see above) and sending e-mail (see below) Panorama can control your default web and e-mail client to view pages and set up outgoing e-mail.

Displaying a Web Page

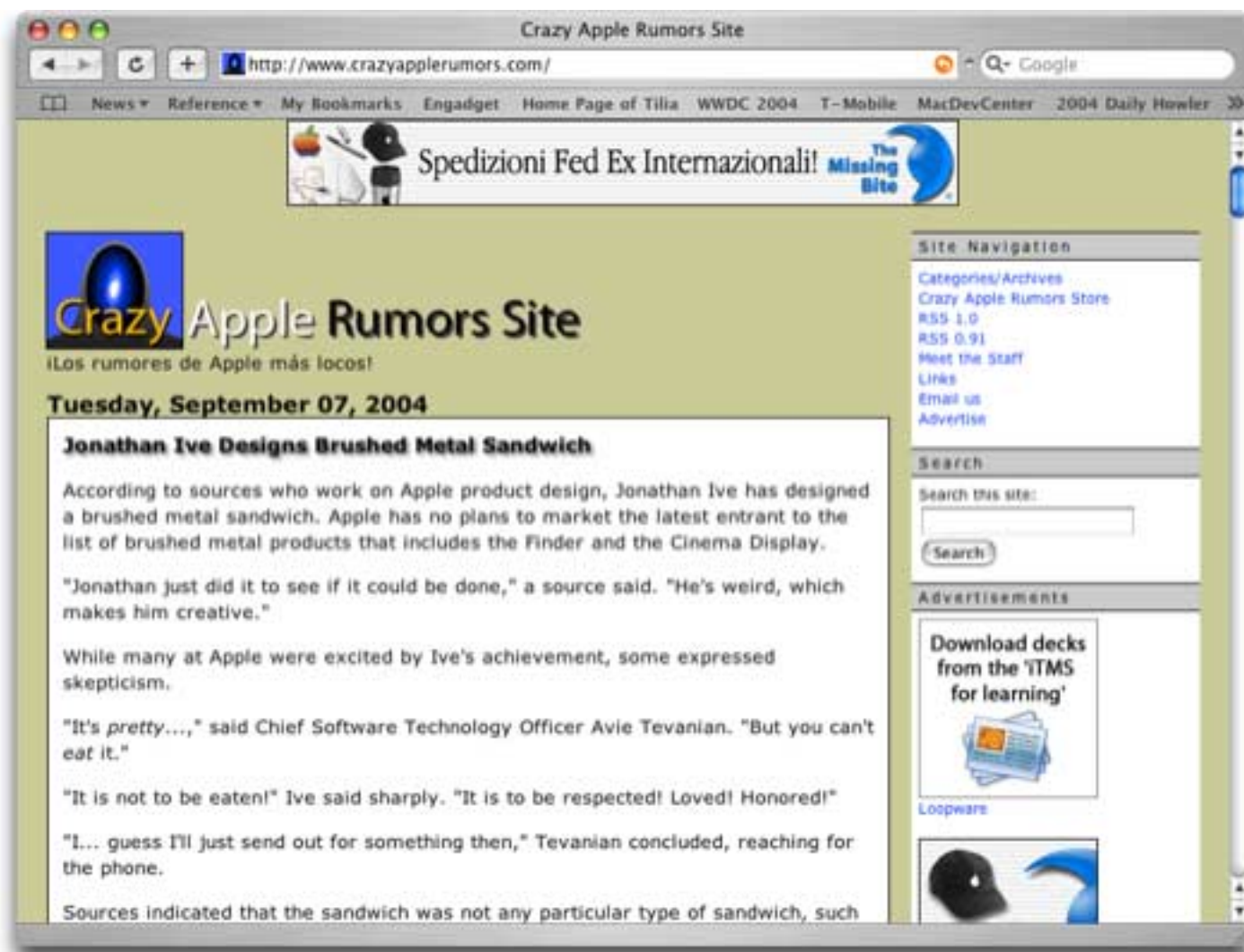
To display a web page use the **shellopendocument** statement.

```
shellopendocument url
```

The **url** must be a complete url, beginning with **http://**.

```
shellopendocument "http://www.crazyapplerumors.com"
```

The HTML document will open in your default browser. On Windows this is normally Internet Explorer, on MacOS X it is normally Safari.



Displaying a Web Page on a Local Hard Drive

To display a web page that is on your local hard drive (not on the web) use the `openanything` statement.

```
openanything folder,file
```

This statement will display an HTML file named `MyLife.html`. This file is located on `My Disk` in the `My Stuff` folder.

```
openanything folder("My Disk:My Stuff:"),"MyLife.html"
```

Displaying a Map

Use the `openwebmap` statement to display the map for any US address.

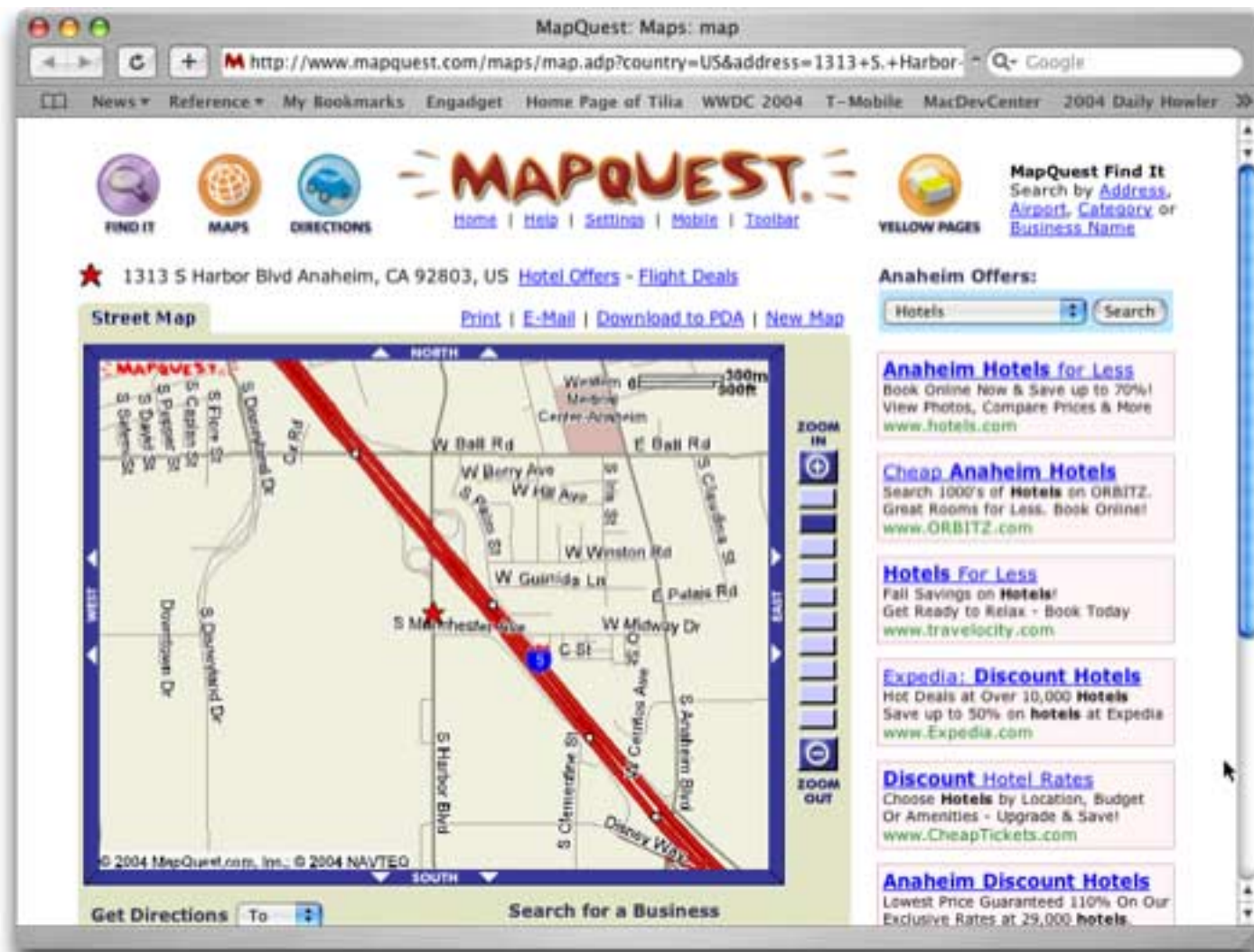
```
openwebmap street,city,state,zip,country
```

The `street`, `city`, `state`, `zip` and `country` parameters specify the address you want to map. Do not include the suite or apartment number in the `street` address. The `country` parameter is currently ignored, this statement only works for United States. Future versions may be more global.

This example opens the default web browser and displays a map showing the location of Disneyland.

```
openwebmap "1313 S. Harbor Blvd. ","Anaheim","CA","92803", ""
```

Here's the resulting map:



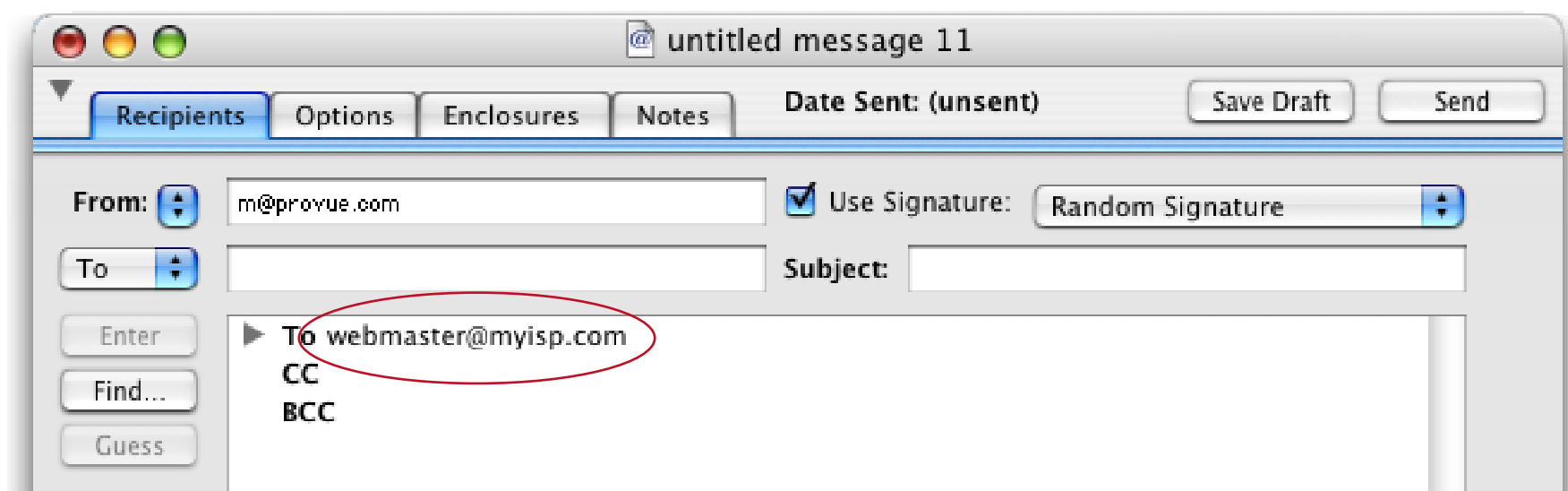
Remember that you can also automatically save this map to a disk file with the `savewebmap` statement (see “[Generating Map Images](#)” on page 617).

Sending an E-Mail

In addition to displaying web pages, the `shellopendocument` statement can also be used to initiate the process of creating a new e-mail. Instead of starting the `url` with `http://`, start it with `mailto:`, followed by the e-mail address. This example creates a new message addressed to `webmaster@myisp.com`.

```
shellopendocument "mailto:webmaster@myisp.com"
```

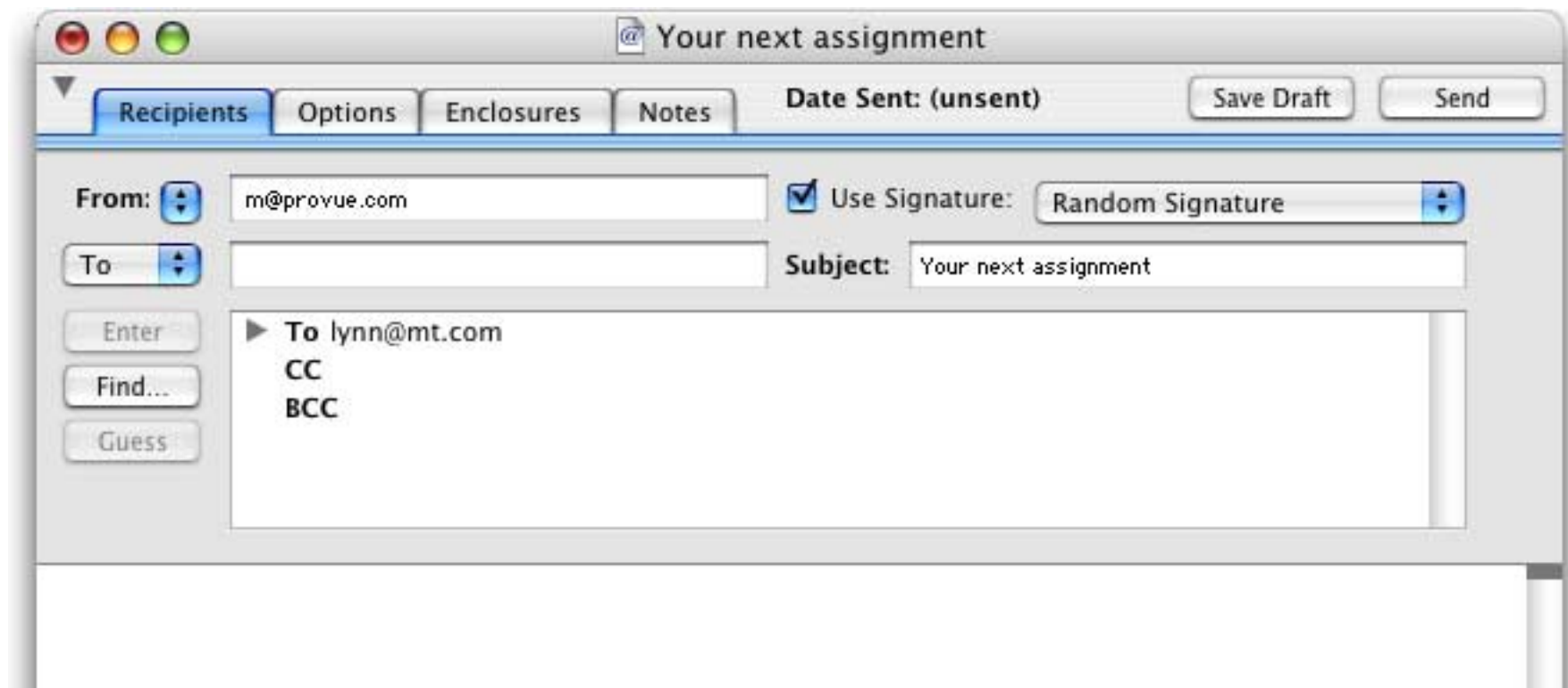
Running a procedure that contains this statement will open your default e-mail client and create a new e-mail message addressed to the specified person.



To specify a subject for the new e-mail, add a suffix beginning with ?. (Note: Not all e-mail clients support this suffix.) This example creates a new message addressed to lynn@mt.com with a subject of **Your next assignment**.

```
shellopendocument "mailto:lynn@mt.com?Your next assignment"
```

Running a procedure that contains this statement will open your default e-mail client and create a new e-mail message addressed to the specified person and with the subject already typed in.



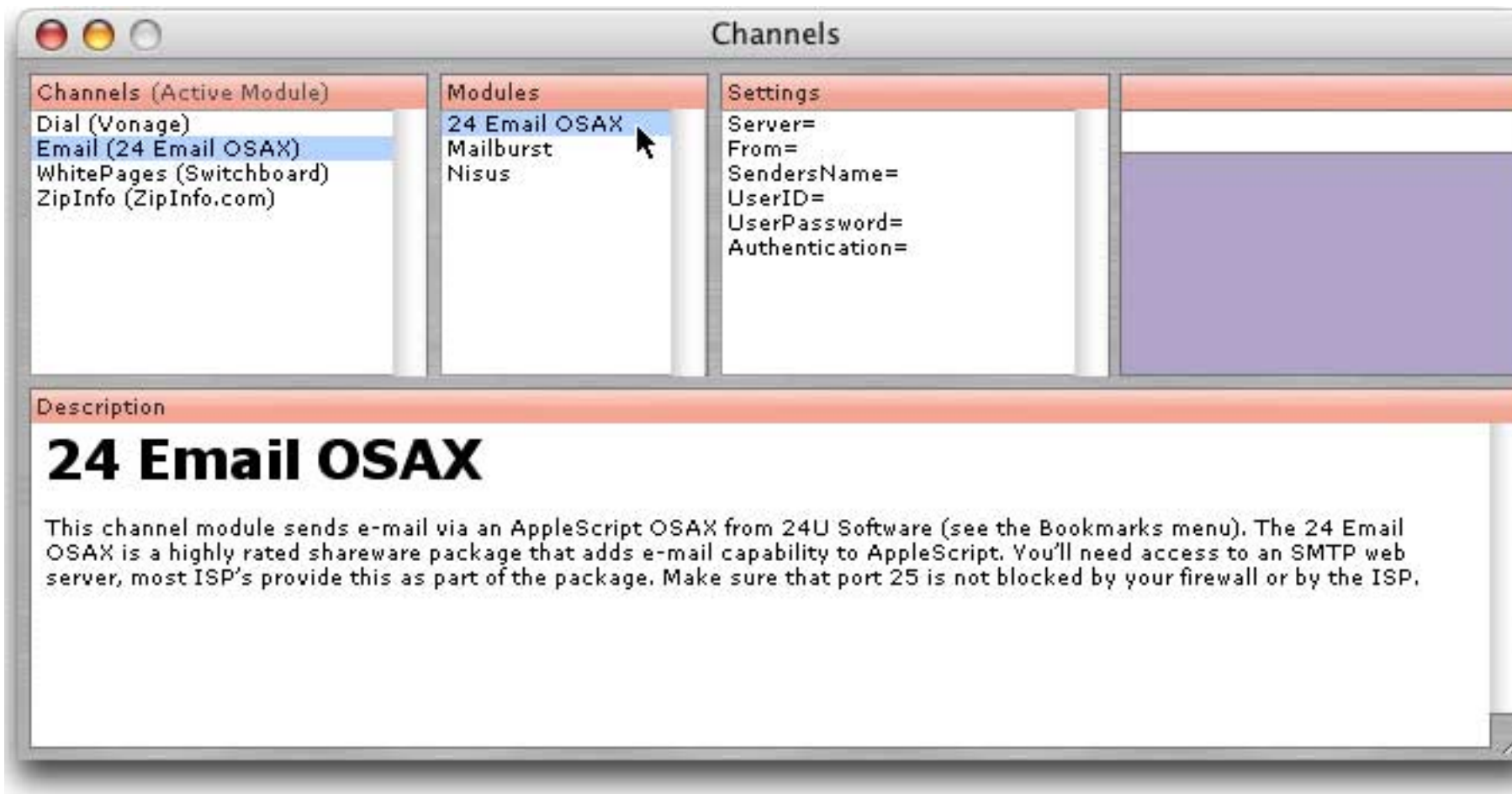
All you need to do is type in the body of the message and press the **Send** button.

Sending E-Mail

Panorama cannot send e-mail by itself. However, through the use of a channel Panorama can interface with external software to send e-mails automatically. Panorama comes with several ready to use channels, and it is also possible to write your own channels.

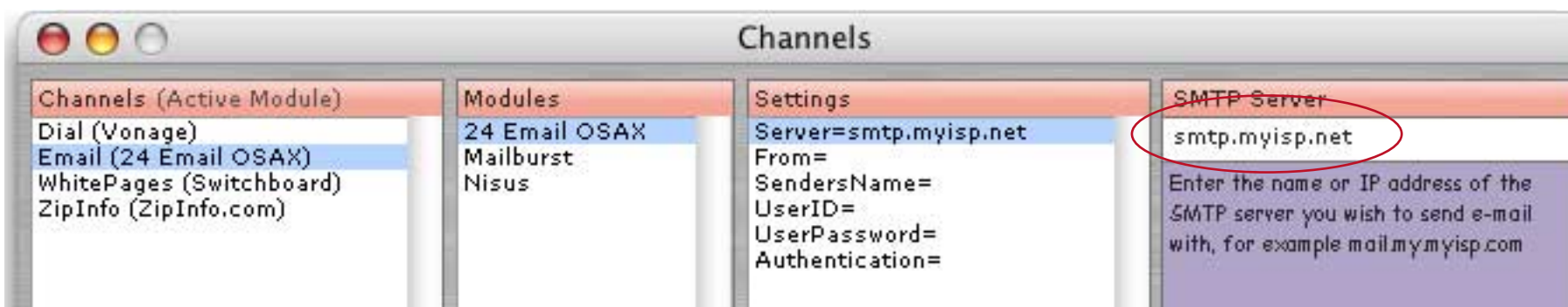
Channel Configuration

Before sending e-mail you'll need to select and configure the e-mail channel you are planning to use. This is done with the **Channels** wizard, which is in the **Preferences** submenu.

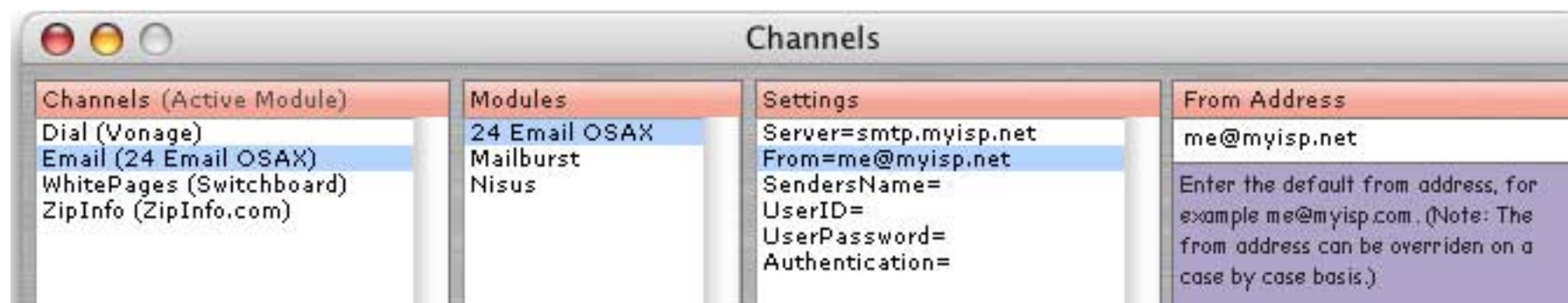


The first step is to click on **E-mail** in the left column, then select the module you want to use. In this case we've picked **24 Email OSAX**, which uses a software package, called, surprisingly enough *24 Email OSAX* (this must be purchased separately from another company). You'll want to pick the module for whatever external software you already have or plan to acquire.

The exact setup details depend on the module you choose. We'll show how to set up for *24 Email OSAX* as an example. Since *24 Email OSAX* communicates directly with an SMTP server, you'll need the same SMTP account information you used when setting up your e-mail client. To start, click on **Server=** and type in the URL for the SMTP server your ISP provides.



Now click on **From=** and type in your e-mail address.



Repeat to type in any additional information required by your ISP, then close the **Channels** wizard when you are done.

Sending a single e-mail

Once you have set up a channel you are ready to send e-mails. To send a single e-mail to a single recipient use the `sendoneemail` statement.

```
sendoneemail from,to,subject,body
```

The **from** parameter is the sender's email address. If blank, the default from address will be used. However, some channels may ignore the from address specified by this channel and always use the default from address.

In addition to the actual email address you can also specify the senders actual name after a comma, for example `joe@acme.net,Joe Smith`. However, not all channels will use this information. It will be ignored if the channel doesn't support this option.

The **to** parameter is the recipient's e-mail address, the address of the person you are sending the e-mail to.

The **subject** parameter is a line of text that will become the subject of the e-mail.

The **body** parameter is the main body of the e-mail message. Panorama only supports text messages, it doesn't support attachments.

This example sends an e-mail to `joe@aol.com`.

```
sendonemail "", "joe@aol.com", "Next Week's meeting",
  "Dear Joe"+¶+¶+"Please bring the new XRF-89 specs with you to the meeting."
```

Sending multiple e-mails

Panorama has two statements for sending multiple copies of the same e-mail to different recipients — `sendbulkemail` and `sendarrayemail`. Use `sendbulkemail` if the recipient e-mail addresses are in a database field. Use `sendarrayemail` if the recipient e-mail addresses are in a text array.

The `sendbulkemail` statement has five parameters.

```
sendbulkemail from,database,recipient,subject,body
```

The **from** parameter is the sender's email address. If blank, the default from address will be used. However, some channels may ignore the from address specified by this channel and always use the default from address.

In addition to the actual email address you can also specify the senders actual name after a comma, for example `joe@acme.net,Joe Smith`. However, not all channels will use this information. It will be ignored if the channel doesn't support this option.

The **database** parameter is the name of the database to extract the e-mail addresses from. Use "" for the current database.

The **recipient** parameter is the name of the field that contains the e-mail addresses you want to send this message to.

The **subject** parameter is a line of text that will become the subject of the e-mail.

The **body** parameter is the main body of the e-mail message. Panorama only supports text messages, it doesn't support attachments.

This example sends an e-mail to every selected person in the **Contacts** database.

```
sendbulkemail "", "Contacts", Email, "Next Week's Meeting",
  "Don't forget, the XRF-89 specification meeting is next Tuesday at 10:30 AM. "+
  "Joe Wilson will be presenting the new specifications."
```

The **sendarrayemail** statement sends multiple e-mails to recipients listed in a comma separated array.

```
sendarrayemail from,to,subject,body
```

The **from** parameter is the sender's email address. If blank, the default from address will be used. However, some channels may ignore the from address specified by this channel and always use the default from address.

In addition to the actual email address you can also specify the sender's actual name after a comma, for example **joe@acme.net, Joe Smith**. However, not all channels will use this information. It will be ignored if the channel doesn't support this option.

The **to** parameter is a comma separated array containing recipient's e-mail addresses.

The **subject** parameter is a line of text that will become the subject of the e-mail.

The **body** parameter is the main body of the e-mail message. Panorama only supports text messages, it doesn't support attachments.

This example sends an e-mail to three recipients: joe, jack and sue.

```
sendarraymail "", "joe@aol.com, jack@earthlink.com, sue@hotmail.net",
  "Next Week's Meeting",
  "Don't forget, the XRF-89 specification meeting is next Tuesday at 10:30 AM. "+
  "Joe Wilson will be presenting the new specifications."
```

Panorama also has one additional statement for sending e-mail to multiple recipients — **sendemail**. This statement is a bit more complicated but gives you more precise control over your e-mail, for example you can send e-mail to some recipients as CC and some as BCC. For more information on using this statement see the **Programming Reference wizard**.

Programming Graphic Objects on the Fly

Graphic objects are usually manipulated manually in Graphics Mode. A procedure can also be programmed to perform manipulations on graphic objects. For example a procedure can move or change the size of objects, change the color of objects, change the font of text objects, etc. When a procedure manipulates graphic objects it does so directly in Data Mode (not in Graphic Mode).

Although procedures can manipulate graphics, they cannot do everything that you can do manually in Graphics Mode. A procedure cannot create new objects or delete existing objects, and it cannot make any change that would change the amount of memory used by the graphic object. (For example a procedure cannot change the text in an auto-wrap text object.)

Basics of Graphic Object Programming

Working with graphic objects is a two step process. First, the program must identify the object or objects that need to be modified. This is called selecting the objects. The process is similar to manually selecting a graphic object by clicking on it or dragging around it. (However, unlike objects that are selected manually in graphics mode, no handles appear at the corners of objects that are selected by a procedure.) Of course a procedure cannot click on an object, so it has to use one or more properties of the object to identify it. For example, you can select an object based on its name, based on its position, based on its color, or based on a number of other attributes (or combinations of attributes).

Once at least one object is selected the procedure can use the `changeobjects` statement to change the object or objects. The `changeobjects` statement can change one property of an object (or objects) at time. If you need to change more than one property (for example position and color) you'll need to use more than one `changeobjects` statement.

Selecting an Object by Name

If an object has a unique name within a form, the simplest way to select the object is using that name. Any graphic object can have a name that can be used to identify that object. To give an object a name first select the object (in Graphics Mode), then use the Object Name command in the Edit menu or click on the object name in the Graphic Control Strip (see "[Object Type/Object Name](#)" on page 533 of the *Panorama Handbook*). (The Graphic Control Strip can also display the name of the object when you click on the object.)

To select an object by name use the `object` statement. This statement has one parameter—the name of the object to select. For example, to select an object named `Swiss Cheese` use this procedure:

```
object "Swiss Cheese"
```

The parameter must match the object name exactly, including upper and lower case. If there is more than one object named `Swiss Cheese` this statement will select the one farthest to the back. (To select multiple objects at a time use the `selectobjects` statement, described in the next section.)

If the user is currently editing using a SuperObject (text editor or word processor) the procedure can find out the name of the object being edited with the `info("editing")` function. You can use this function with the `object` statement to select the object, and possibly change one or more of its attributes. (Note: This function scans all the objects in the current form, so if you are going to use it over and over again it might be faster to use it once and copy the name into a local variable, then use the local variable.)

Selecting Multiple Objects

To select multiple objects at once use the `selectobjects` statement. This statement scans the objects in the current form and selects some of them based on a formula. The formula can use the `objectinfo()` function (see the next section) to examine each object as it is scanned and decide whether or not the object should be selected. For example, the statement below scans the current form and selects all objects that are pure blue.

```
selectobjects objectinfo("color")=rgb(0,0,65535)
```

To quickly select all of the objects in the current form, use the `selectallobjects` statement. To quickly unselect all of the objects in the current form, use the `selectnoobjects` statement.

No matter how the objects are selected, they will remain selected until you close the form, switch the form into graphics mode, or perform another statement that selects objects. Once one or more objects are selected you can use the `changeobjects` statement to change many of the attributes of the object (more on this later in this chapter).

Getting Information About Individual Objects

A procedure does not have eyes to see the graphic objects in a form. Instead of eyes or a camera, the procedure uses the `objectinfo` function to gather information about graphic objects. The `objectinfo` function has one parameter—the type of information you want to collect (object location, size, color, font, etc.).

Like a camera, the `objectinfo` function must be “pointed” at a specific object. There are several statements that can “point” at a specific object, including the `object` statement and the `selectobjects` statement (see previous sections).

Here is an example of the object statement and `objectinfo` function in action. This example finds out the font and text size of the object named `MySpecialButton`.

```
local myFont, mySize
object "MySpecialButton"
if info("found")
  myFont=objectinfo("font")
  mySize=objectinfo("textsize")
endif
```

There are about a dozen types of information the `objectinfo` function can extract from an object.

objectinfo(function	Description
<code>objectinfo("rectangle")</code>	<p>This option returns the dimensions (location and size) of the object. The dimensions are returned using the rectangle data type (see “Rectangles” on page 149). The rectangle is returned in form relative co-ordinates (see “XYTOXY()” on page 5910).</p> <p>The example below selects the data cell(s) the user clicked on. The procedure uses the <code>inrectangle</code> function to determine which object (if any) was clicked on. (Note: Presumably this procedure would be triggered by a push button which covers the data cell objects.)</p> <pre>local hitPt, hitField hitPt=xytoxy(info("click"), "Screen", "Form") selectobjects inrectangle(hitPt,objectinfo("rectangle")) and objectinfo("type") beginswith "Data Cell:" objectnumber 1 hitField=objectinfo("type")[":",-1][-2,-1] if hitField="" stop endif field hitField editcell</pre> <p>If the user did click on a data cell, the procedure activates the cell.</p>

objectinfo(function	Description
<p>objectinfo("name")</p>	<p>This option returns the name of the object. This is the name that is assigned by the Object Name dialog (in the Edit menu, or Graphic Control Strip). The two lines shown below are basically equivalent.</p> <pre>object "Swiss Cheese" selectobjects objectinfo("name")="Swiss Cheese"</pre> <p>These two statements are not completely equivalent. If there is more than one object named Swiss Cheese the selectobjects statement will select all of them. The object statement will select only the one closest to the back.</p> <p>The objectinfo("name") function can be used in a formula to decode object names. For example, if a form contains rows and columns you can give each cell a name like c1r1, c1r2, ... c4r12. Using the objectinfo("name") function a procedure could decode these names and select a specific column or row. For example, here is a procedure that selects the third column:</p> <pre>selectobjects objectinfo("name")[1,2]="c3"</pre> <p>Here is another procedure that selects the seventh row:</p> <pre>selectobjects objectinfo("name")[3,4]="r7"</pre> <p>By carefully assigning object names you can often simplify the design of your procedures tremendously. Look for patterns that you can take advantage of.</p>
<p>objectinfo("type")</p>	<p>This option returns the type of the object. The object types are:</p> <pre>Rectangle Rounded Rectangle Oval Line Picture Auto-Wrap Text Click Text Data Cell:<field> Button Chart Flash Art Flash Sound Balloon Help SuperObject:<type of SuperObject> Tile:<type of tile> Group</pre> <p>To see a complete list of SuperObject types see the objectinfo("custom") function (Page 639). To see a complete list of tile types see the objectinfo("tile") function (Page 638).</p> <p>Here is a procedure that uses this function to select all of the rectangles in the current form.</p> <pre>selectobjects objectinfo("type")="Rectangle"</pre>
<p>objectinfo("font")</p>	<p>This option returns the font for this object. If the option does not have a font (an oval, for example) this option will return empty text.</p> <p>This procedure converts all Courier text to American Typewriter.</p> <pre>selectobjects objectinfo("font")="Courier" changeobjects "font", "American Typewriter"</pre>

objectinfo(function	Description
objectinfo("textsize")	<p>This option returns the size of the text displayed by this object. If the object does not have a text size (an oval, for example) this option will return zero. Here is a procedure that selects all objects with a text size greater than 18 points (1/4 inch) and changes them to American Typewriter.</p> <pre data-bbox="891 483 1699 554">selectobjects objectinfo("textsize")>18 changeobjects "font","American Typewriter"</pre>
objectinfo("textstyle")	<p>This option returns the text style of text displayed by the object. The text style is a number that is created by adding up the numbers for each individual style from the table below. For example, for bold italic text the style will be 3.</p> <pre data-bbox="891 800 1122 1020">0 Plain 1 Bold 2 Italic 4 Underline 8 Outline 16 Shadow</pre> <p>The example below selects all italic objects and then changes the color of the italic objects to blue.</p> <pre data-bbox="891 1170 1720 1241">selectobjects objectinfo("textstyle") and 2 changeobjects "color",rgb(0,0,65535)</pre>
objectinfo("color")	<p>This option returns the color of the object (see “Colors” on page 154). For example, this procedure selects all objects with brightness below 50%, then changes it to a minimum brightness of 50%.</p> <pre data-bbox="891 1450 1873 1705">selectobjects brightness(objectinfo("color"))<32768 changeobjects "color", hsb(hue(objectinfo("color")), saturation(objectinfo("color")), 32768)</pre>
objectinfo("selected")	<p>This option returns whether or not the object is already selected (by a previous selectobjects statement).</p>
objectinfo("locked")	<p>This option returns true or false depending on whether or not the object is locked. (A locked object cannot be modified when in graphic editing mode, see “Locked Objects” on page 575 of the <i>Panorama Handbook</i>.) The example below selects all rectangles that are not locked.</p> <pre data-bbox="891 2050 1738 2121">selectobjects objectinfo("type")="Rectangle" and not objectinfo("locked")</pre>
objectinfo("expandable")	<p>This option returns true or false depending on whether or not the object can expand depending on the amount of data to be printed (see “Variable Height Records” on page 1123 of the <i>Panorama Handbook</i>).</p>
objectinfo("expandshrink")	<p>This option returns true or false depending on whether or not the object can expand or shrink depending on the amount of data to be printed (see “The Expand/Shrink Option” on page 1130 of the <i>Panorama Handbook</i>).</p>

objectinfo(function	Description																				
objectinfo("text")	<p>This option returns the text in auto-wrap text objects or click text objects (see “Fixed Text Objects” on page 587 of the <i>Panorama Handbook</i>). When used with any other type of object it returns empty text.</p> <p>This example changes all text objects that contain the word Phone to italic.</p> <pre>selectobjects objectinfo("text") contains "Phone" changeobjects "textstyle", objectinfo("textstyle") or 2</pre>																				
objectinfo("fillpattern")	<p>This option returns the fill pattern of the object (if any, see “Fill Pattern” on page 521 of the <i>Panorama Handbook</i>). Patterns are 8 bytes of raw data (see “Raw Binary Data” on page 156). Here are some formulas for typical patterns.</p> <table border="0"> <thead> <tr> <th data-bbox="891 856 1028 885"><u>Formula</u></th> <th data-bbox="1450 856 1587 885"><u>Pattern</u></th> </tr> </thead> <tbody> <tr> <td data-bbox="891 913 1428 941">radix(16,"FFFFFFFFFFFFFFFF")</td> <td data-bbox="1450 913 1546 941">black</td> </tr> <tr> <td data-bbox="891 950 1393 978">radix(16,"0000000000000000")</td> <td data-bbox="1450 950 1546 978">white</td> </tr> <tr> <td data-bbox="891 986 926 1015">"</td> <td data-bbox="1450 986 1793 1015">none (transparent)</td> </tr> <tr> <td data-bbox="891 1023 1428 1052">radix(16,"AA55AA55AA55AA55")</td> <td data-bbox="1450 1023 1760 1052">50% gray pattern</td> </tr> <tr> <td data-bbox="891 1060 1428 1088">radix(16,"8822882288228822")</td> <td data-bbox="1450 1060 1646 1088">light gray</td> </tr> <tr> <td data-bbox="891 1097 1428 1125">radix(16,"DD77DD77DD77DD77")</td> <td data-bbox="1450 1097 1624 1125">dark gray</td> </tr> <tr> <td data-bbox="891 1134 1428 1162">radix(16,"8888888888888888")</td> <td data-bbox="1450 1134 1720 1162">vertical lines</td> </tr> <tr> <td data-bbox="891 1170 1428 1199">radix(16,"FF000000FF000000")</td> <td data-bbox="1450 1170 1760 1199">horizontal lines</td> </tr> <tr> <td data-bbox="891 1207 1428 1235">radix(16,"FF888888FF888888")</td> <td data-bbox="1450 1207 1664 1235">cross-hatch</td> </tr> </tbody> </table> <p>This list shows only a few of the possible patterns—there are literally millions of patterns that can be created.</p>	<u>Formula</u>	<u>Pattern</u>	radix(16,"FFFFFFFFFFFFFFFF")	black	radix(16,"0000000000000000")	white	"	none (transparent)	radix(16,"AA55AA55AA55AA55")	50% gray pattern	radix(16,"8822882288228822")	light gray	radix(16,"DD77DD77DD77DD77")	dark gray	radix(16,"8888888888888888")	vertical lines	radix(16,"FF000000FF000000")	horizontal lines	radix(16,"FF888888FF888888")	cross-hatch
<u>Formula</u>	<u>Pattern</u>																				
radix(16,"FFFFFFFFFFFFFFFF")	black																				
radix(16,"0000000000000000")	white																				
"	none (transparent)																				
radix(16,"AA55AA55AA55AA55")	50% gray pattern																				
radix(16,"8822882288228822")	light gray																				
radix(16,"DD77DD77DD77DD77")	dark gray																				
radix(16,"8888888888888888")	vertical lines																				
radix(16,"FF000000FF000000")	horizontal lines																				
radix(16,"FF888888FF888888")	cross-hatch																				
objectinfo("linepattern")	<p>This option returns the line pattern of the object (if any, see “Line Pattern” on page 523 of the <i>Panorama Handbook</i>). Patterns are 8 bytes of raw data (see “Raw Binary Data” on page 156). Here are some formulas for typical patterns.</p> <table border="0"> <thead> <tr> <th data-bbox="891 1572 1028 1600"><u>Formula</u></th> <th data-bbox="1450 1572 1587 1600"><u>Pattern</u></th> </tr> </thead> <tbody> <tr> <td data-bbox="891 1628 1428 1657">radix(16,"FFFFFFFFFFFFFFFF")</td> <td data-bbox="1450 1628 1546 1657">black</td> </tr> <tr> <td data-bbox="891 1665 1393 1693">radix(16,"0000000000000000")</td> <td data-bbox="1450 1665 1546 1693">white</td> </tr> <tr> <td data-bbox="891 1702 926 1730">"</td> <td data-bbox="1450 1702 1793 1730">none (transparent)</td> </tr> <tr> <td data-bbox="891 1739 1428 1767">radix(16,"AA55AA55AA55AA55")</td> <td data-bbox="1450 1739 1760 1767">50% gray pattern</td> </tr> <tr> <td data-bbox="891 1775 1428 1804">radix(16,"8822882288228822")</td> <td data-bbox="1450 1775 1646 1804">light gray</td> </tr> <tr> <td data-bbox="891 1812 1428 1841">radix(16,"DD77DD77DD77DD77")</td> <td data-bbox="1450 1812 1624 1841">dark gray</td> </tr> <tr> <td data-bbox="891 1849 1428 1877">radix(16,"8888888888888888")</td> <td data-bbox="1450 1849 1720 1877">vertical lines</td> </tr> <tr> <td data-bbox="891 1886 1428 1914">radix(16,"FF000000FF000000")</td> <td data-bbox="1450 1886 1760 1914">horizontal lines</td> </tr> <tr> <td data-bbox="891 1923 1428 1951">radix(16,"FF888888FF888888")</td> <td data-bbox="1450 1923 1664 1951">cross-hatch</td> </tr> </tbody> </table> <p>This list shows only a few of the possible patterns—there are literally millions of patterns that can be created</p>	<u>Formula</u>	<u>Pattern</u>	radix(16,"FFFFFFFFFFFFFFFF")	black	radix(16,"0000000000000000")	white	"	none (transparent)	radix(16,"AA55AA55AA55AA55")	50% gray pattern	radix(16,"8822882288228822")	light gray	radix(16,"DD77DD77DD77DD77")	dark gray	radix(16,"8888888888888888")	vertical lines	radix(16,"FF000000FF000000")	horizontal lines	radix(16,"FF888888FF888888")	cross-hatch
<u>Formula</u>	<u>Pattern</u>																				
radix(16,"FFFFFFFFFFFFFFFF")	black																				
radix(16,"0000000000000000")	white																				
"	none (transparent)																				
radix(16,"AA55AA55AA55AA55")	50% gray pattern																				
radix(16,"8822882288228822")	light gray																				
radix(16,"DD77DD77DD77DD77")	dark gray																				
radix(16,"8888888888888888")	vertical lines																				
radix(16,"FF000000FF000000")	horizontal lines																				
radix(16,"FF888888FF888888")	cross-hatch																				
objectinfo("linewidth")	<p>This option returns the line width of the object (if any, see “Line Width” on page 525 of the <i>Panorama Handbook</i>). The line width is a number from 1 to 8, or zero if this object does not support a line width.</p>																				

objectinfo(function	Description
objectinfo("tile")	<p>This option returns the type of tile (if the object is a tile, otherwise it returns ""). You can also get this information using the objectinfo("type") function. The tile type will be one of the names in this list.</p> <p>"1st page Header" "1st page Header (Center) " "1st page Header (Right) " "Header" "Header (Center) " "Header (Right) " "Table Header " "Group Header (1) " "Group Header (2) " "Group Header (3) " "Group Header (4) " "Group Header (5) " "Group Header (6) " "Group Header (7) " "Group Sidebar (1) " "Group Sidebar (2) " "Group Sidebar (3) " "Group Sidebar (4) " "Group Sidebar (5) " "Group Sidebar (6) " "Group Sidebar (7) " "Data " "Summary (1) " "Summary (2) " "Summary (3) " "Summary (4) " "Summary (5) " "Summary (6) " "Summary (7) " "Table Footer " "Footer" "Footer (Center) " "Footer (Right) " "Left Margin " "Right Margin " "Backdrop " "Spacer " "Data (Page 2) " "Data (Page 3) " "Data (Page 4) " "Data (Page 5) " "Data (Page 6) " "Data (Page 7) " "Data (Page 8) " "Data (Page 9) " "Top Margin" "Data Overflow" "Last page Header" "Last page Header (Center) " "Last page Header (Right) "</p> <p>For more information on report tiles see “Working with Tiles” on page 1062 of the <i>Panorama Handbook</i>.</p>

objectinfo(function	Description
objectinfo("custom")	<p>This option only works with SuperObjects. It returns the type of Super-Object (see list below). This information can also be obtained by using the <code>objectinfo("type")</code> function.</p> <pre> "Text Display" "Text Editor" "PgCell" (word processor) "Super Flash Art" "Push Button" "Flash Art Push Button" "Data Button" "Sticky Push Button" "Flash Art Data Button" "PopUp Menu" "Text List" (scrollable list) "Scroll Bar" "Super Matrix" "Auto Grow" (elastic form) </pre>
objectinfo("ID")	<p>This option returns a unique number that can be used to identify this object later. The number is valid as long as the form is not edited in graphics mode. The <code>objectid</code> statement can use this unique ID number to re-locate this object later (see "Object ID Values" on page 643).</p>
objectinfo("count")	<p>This option applies not to a specific object, but to the entire form. It counts the number of currently selected objects. For example, this example displays the number of rectangles in the current form.</p> <pre> selectobjects objectinfo("type") contains "rectangle" message "This form contains "+ str(objectinfo("count"))+ " rectangles." </pre>
objectinfo("boundary")	<p>This option applies not to a specific object, but to the entire form. It calculates the minimum rectangle that encloses all of the selected objects.</p>

Modifying Selected Objects

A program can use the `changeobjects` statement to modify certain attributes of selected objects. The `changeobjects` statement has two parameters:

```
changeobjects how,data
```

The `how` parameter specifies how the objects should be adjusted—a new font, a new color, new position, etc. The `data` parameter specifies the new object attributes—"Palatino", `rgb(5000,12000,48000)`, `rectangle(100,120,410,240)`, etc. The following table describes each of the options available.

Option	Description
rectangle	<p>This option changes the rectangle of all selected objects. This example moves all selected objects down and to the right by 36 pixels (1/2 inch).</p> <pre> changeobjects "rectangle", rectangleadjust(objectinfo("rectangle"), 36,36,36,36) </pre>

Option	Description
fieldname	<p>This option applies only to data cells. It changes the field associated with the any selected data cells. The example below changes all Qty cells to Price cells (Qty1 to Price1, Qty2 to Price2, etc.)</p> <pre data-bbox="556 376 1517 480">selectobjects objectinfo("fieldname") match "Qty?" changeobjects "fieldname", "Price"+objectinfo("fieldname")[4,-1]</pre>
font	<p>This option changes the font of selected objects. Non text objects will not be affected. The example below sets the font of all data cells to Times Roman.</p> <pre data-bbox="556 729 1408 800">selectobjects objectinfo("type")="Data Cell" changeobjects "font","Times Roman"</pre>
textsize	<p>This option changes the text size of selected objects. Non text objects will not be affected. The example below reduces the text size of all data cells by 3 points, down to a minimum of 9 points.</p> <pre data-bbox="556 970 1408 1074">selectobjects objectinfo("type")="Data Cell" changeobjects "textsize", maximum(9,objectinfo("textsize")-3)</pre>
textstyle	<p>This option changes the text size of selected objects. Non text objects will not be affected. The text style is a number that is created by adding up the numbers for each individual style from the table below. For example, for bold italic text the style will be 3.</p> <pre data-bbox="556 1286 797 1507">0 Plain 1 Bold 2 Italic 4 Underline 8 Outline 16 Shadow</pre> <p>The example below sets the style of all data cells to bold italic.</p> <pre data-bbox="556 1620 1408 1691">selectobjects objectinfo("type")="Data Cell" changeobjects "textstyle",3</pre>
color	<p>This option changes the color of the selected objects (see “Colors” on page 154). The example procedure below changes any pure red objects on the current form into blue objects.</p> <pre data-bbox="556 1860 1474 1931">selectobjects objectinfo("color")=rgb(65535,0,0) changeobjects "color",rgb(0,0,65535)</pre>

Option	Description																				
fillpattern	<p>This option changes the fill pattern of the selected objects (see “Fill Pattern” on page 521 of the <i>Panorama Handbook</i>). Patterns are 8 bytes of raw data (see “Raw Binary Data” on page 156). Here are some formulas for typical patterns.</p> <table border="0"> <thead> <tr> <th data-bbox="559 410 694 438"><u>Formula</u></th> <th data-bbox="1116 410 1251 438"><u>Pattern</u></th> </tr> </thead> <tbody> <tr> <td data-bbox="559 469 1092 497"><code>radix(16,"FFFFFFFFFFFFFFFF")</code></td> <td data-bbox="1116 469 1214 497">black</td> </tr> <tr> <td data-bbox="559 506 1057 534"><code>radix(16,"0000000000000000")</code></td> <td data-bbox="1116 506 1214 534">white</td> </tr> <tr> <td data-bbox="559 542 591 571">""</td> <td data-bbox="1116 542 1458 571">none (transparent)</td> </tr> <tr> <td data-bbox="559 579 1092 608"><code>radix(16,"AA55AA55AA55AA55")</code></td> <td data-bbox="1116 579 1426 608">50% gray pattern</td> </tr> <tr> <td data-bbox="559 616 1092 644"><code>radix(16,"8822882288228822")</code></td> <td data-bbox="1116 616 1312 644">light gray</td> </tr> <tr> <td data-bbox="559 653 1092 681"><code>radix(16,"DD77DD77DD77DD77")</code></td> <td data-bbox="1116 653 1297 681">dark gray</td> </tr> <tr> <td data-bbox="559 690 1092 718"><code>radix(16,"8888888888888888")</code></td> <td data-bbox="1116 690 1384 718">vertical lines</td> </tr> <tr> <td data-bbox="559 726 1092 755"><code>radix(16,"FF000000FF000000")</code></td> <td data-bbox="1116 726 1421 755">horizontal lines</td> </tr> <tr> <td data-bbox="559 763 1092 791"><code>radix(16,"FF888888FF888888")</code></td> <td data-bbox="1116 763 1327 791">cross-hatch</td> </tr> </tbody> </table> <p>This list shows only a few of the possible patterns—there are literally millions of patterns that can be created. The example procedure below sets the Check Background object to a dark gray pattern.</p> <pre>selectobjects objectinfo("name")="Check Background" changeobjects "fillpattern",radix(16,"DD77DD77DD77DD77")</pre>	<u>Formula</u>	<u>Pattern</u>	<code>radix(16,"FFFFFFFFFFFFFFFF")</code>	black	<code>radix(16,"0000000000000000")</code>	white	""	none (transparent)	<code>radix(16,"AA55AA55AA55AA55")</code>	50% gray pattern	<code>radix(16,"8822882288228822")</code>	light gray	<code>radix(16,"DD77DD77DD77DD77")</code>	dark gray	<code>radix(16,"8888888888888888")</code>	vertical lines	<code>radix(16,"FF000000FF000000")</code>	horizontal lines	<code>radix(16,"FF888888FF888888")</code>	cross-hatch
<u>Formula</u>	<u>Pattern</u>																				
<code>radix(16,"FFFFFFFFFFFFFFFF")</code>	black																				
<code>radix(16,"0000000000000000")</code>	white																				
""	none (transparent)																				
<code>radix(16,"AA55AA55AA55AA55")</code>	50% gray pattern																				
<code>radix(16,"8822882288228822")</code>	light gray																				
<code>radix(16,"DD77DD77DD77DD77")</code>	dark gray																				
<code>radix(16,"8888888888888888")</code>	vertical lines																				
<code>radix(16,"FF000000FF000000")</code>	horizontal lines																				
<code>radix(16,"FF888888FF888888")</code>	cross-hatch																				
linepattern	<p>This option changes the line pattern of the selected objects (see “Line Pattern” on page 523 of the <i>Panorama Handbook</i>). Patterns are 8 bytes of raw data (see “Raw Binary Data” on page 156). Here are some formulas for typical patterns.</p> <table border="0"> <thead> <tr> <th data-bbox="559 1238 694 1266"><u>Formula</u></th> <th data-bbox="1116 1238 1251 1266"><u>Pattern</u></th> </tr> </thead> <tbody> <tr> <td data-bbox="559 1298 1092 1326"><code>radix(16,"FFFFFFFFFFFFFFFF")</code></td> <td data-bbox="1116 1298 1214 1326">black</td> </tr> <tr> <td data-bbox="559 1334 1057 1363"><code>radix(16,"0000000000000000")</code></td> <td data-bbox="1116 1334 1214 1363">white</td> </tr> <tr> <td data-bbox="559 1371 591 1399">""</td> <td data-bbox="1116 1371 1458 1399">none (transparent)</td> </tr> <tr> <td data-bbox="559 1408 1092 1436"><code>radix(16,"AA55AA55AA55AA55")</code></td> <td data-bbox="1116 1408 1426 1436">50% gray pattern</td> </tr> <tr> <td data-bbox="559 1445 1092 1473"><code>radix(16,"8822882288228822")</code></td> <td data-bbox="1116 1445 1312 1473">light gray</td> </tr> <tr> <td data-bbox="559 1481 1092 1510"><code>radix(16,"DD77DD77DD77DD77")</code></td> <td data-bbox="1116 1481 1297 1510">dark gray</td> </tr> <tr> <td data-bbox="559 1518 1092 1546"><code>radix(16,"8888888888888888")</code></td> <td data-bbox="1116 1518 1384 1546">vertical lines</td> </tr> <tr> <td data-bbox="559 1555 1092 1583"><code>radix(16,"FF000000FF000000")</code></td> <td data-bbox="1116 1555 1421 1583">horizontal lines</td> </tr> <tr> <td data-bbox="559 1592 1092 1620"><code>radix(16,"FF888888FF888888")</code></td> <td data-bbox="1116 1592 1327 1620">cross-hatch</td> </tr> </tbody> </table> <p>This list shows only a few of the possible patterns—there are literally millions of patterns that can be created. The example procedure below sets the Check Background object to a 50% gray pattern (which will display a dotted line).</p> <pre>selectobjects objectinfo("name")="Check Border" changeobjects "linepattern",radix(16,"AA55AA55AA55AA55")</pre>	<u>Formula</u>	<u>Pattern</u>	<code>radix(16,"FFFFFFFFFFFFFFFF")</code>	black	<code>radix(16,"0000000000000000")</code>	white	""	none (transparent)	<code>radix(16,"AA55AA55AA55AA55")</code>	50% gray pattern	<code>radix(16,"8822882288228822")</code>	light gray	<code>radix(16,"DD77DD77DD77DD77")</code>	dark gray	<code>radix(16,"8888888888888888")</code>	vertical lines	<code>radix(16,"FF000000FF000000")</code>	horizontal lines	<code>radix(16,"FF888888FF888888")</code>	cross-hatch
<u>Formula</u>	<u>Pattern</u>																				
<code>radix(16,"FFFFFFFFFFFFFFFF")</code>	black																				
<code>radix(16,"0000000000000000")</code>	white																				
""	none (transparent)																				
<code>radix(16,"AA55AA55AA55AA55")</code>	50% gray pattern																				
<code>radix(16,"8822882288228822")</code>	light gray																				
<code>radix(16,"DD77DD77DD77DD77")</code>	dark gray																				
<code>radix(16,"8888888888888888")</code>	vertical lines																				
<code>radix(16,"FF000000FF000000")</code>	horizontal lines																				
<code>radix(16,"FF888888FF888888")</code>	cross-hatch																				
linewidth	<p>This option changes the line width of the selected objects (see “Line Width” on page 525 of the <i>Panorama Handbook</i>). The example below sets the line width of the Check Background object to 4 pixels.</p> <pre>selectobjects objectinfo("name")="Check Border" changeobjects "linewidth",4</pre>																				
expandable	<p>This option allows a procedure to make an object expandable (so that it will expand when printed in a custom report, see “Variable Height Records” on page 1123 of the <i>Panorama Handbook</i>). Use <code>-1</code> to make the selected objects expandable and <code>0</code> to make the objects fixed height. This example makes every auto-wrap text object on the current form expandable.</p> <pre>selectobjects objectinfo("type")="Auto-Wrap Text" changeobjects "expandable",-1</pre>																				

Option	Description
expandshrink	<p>This option allows a procedure to make an object expandable/shrinkable (so that it will expand or shrink as necessary when printed in a custom report, see “The Expand/Shrink Option” on page 1130 of the <i>Panorama Handbook</i>). Use -1 to make the selected objects expand/shrinkable and 0 to make the objects fixed height. This example makes every auto-wrap text object on the current form expand/shrinkable.</p> <pre>selectobjects objectinfo("type")="Auto-Wrap Text" changeobjects "expandshrink",-1</pre>
lock	<p>This option can lock or unlock a graphic object (see “Locked Objects” on page 575 of the <i>Panorama Handbook</i>). Use -1 to lock the selected objects and 0 to unlock the objects. This example locks every object on the form.</p> <pre>selectallobjects changeobjects "lock",-1</pre>

The `changeobjects` statement is designed to work closely with the `selectobjects` statement and the `objectinfo()` function. See the previous section (“[Modifying Selected Objects](#)” on page 639) for several additional examples of how these statements can work together.

Getting Information About Selected Objects

The `selectobjects` statement can select dozens or even hundreds of graphic objects. To get information about one of these objects use the `objectnumber` statement. This statement has one parameter, a number which specifies which selected object you want to get information about. After the `objectnumber` statement the procedure should have one or more assignment statements that use the `objectinfo()` function to get information about the object.

Suppose there are 5 objects selected. To find out the name of the first selected object (closest to the back) use the procedure:

```
local objName
objectnumber 1
objName=objectinfo("name")
```

To find out the name of the last selected object (closest to the front) use the procedure:

```
local objName
objectnumber 5
objName=objectinfo("name")
```

If there are not enough selected objects to fulfill the request, the `info("found")` function will return false. In other words, if there are only 3 objects selected and you try to get information about number 7, `info("found")` will be set to false. The procedure below takes advantage of this feature to build a list of all the names of all the SuperObjects in the current form.

```
local objectNames,X
X=1
objectselect objectinfo("type") beginswith "SuperObject"
loop
  objectnumber X
  stoploopif (not info("found"))
  objectNames=sandwich(" ",objectNames,¶)+objectinfo("name")
  X=X+1
while forever
```

You can also use the `objectinfo("count")` function to find out how many objects are selected. Here is another procedure that does the same job as the last example but in a slightly different way.

```
local objectNames,maxObject,X
X=1
objectselect objectinfo("type") beginswith "SuperObject"
maxObject=objectinfo("count")
loop
  stoploopif X>maxObject
  objectnumber X
  objectNames=sandwich(" ",objectNames,¶)+objectinfo("name")
  X=X+1
while forever
```

The example below finds the name of the top object the user clicked on. The procedure uses the `inrectangle()` function to determine which object (if any) was clicked on. (Note: Presumably this procedure would be triggered by a transparent push button which covers all the other objects. This button is not counted as the object the user clicked on.)

```
local hitPt, hitObject
hitPt=xytoxy(info("click"),"Screen","Form")
selectobjects inrectangle(hitPt,objectinfo("rectangle")) and
  objectinfo("type") ≠ "Button"
objectnumber objectinfo("count")
hitObject=objectinfo("name")
```

Object ID Values

Each graphic object has a unique ID value that can be used to identify that object. The ID value is a number that is guaranteed to be unique for that object only. (However, if you edit the form in graphic editing mode the ID value may change.)

A procedure can use the `objectinfo("ID")` function to find out the ID of an object. The procedure can store the ID value and later use it with the `objectid` statement to re-select the object. For example, here is a procedure that finds and stores the ID of an object the user clicks on (see previous section for more details on this example.)

```
global hitObject
local hitPt
hitPt=xytoxy(info("click"),"Screen","Form")
selectobjects inrectangle(hitPt,objectinfo("rectangle")) and
  objectinfo("type") ≠ "Button"
objectnumber objectinfo("count")
hitObject=objectinfo("ID")
```

Later another procedure can re-select this object with a single statement.

```
objectid hitObject
```

You can also use the object ID value to determine the relative front-to-back order of two or more objects. Objects that are closer to the front will have higher ID values, while objects that are closer to the back will have lower ID values.

Redrawing an Object

It's usually not necessary to explicitly redraw an object (or objects), but if it is necessary you can do so with the `drawobjects` statement. This statement has no parameters, and must be preceded by the `object`, `selectobjects`, or `objectid` statements. This example redraws the object called `Swiss Cheese` (see "[Selecting an Object by Name](#)" on page 633).

```
object "Swiss Cheese"
drawobjects
```

This example redraws all of the objects in the current form that are displayed in the font **Courier** (see “[Selecting Multiple Objects](#)” on page 633).

```
selectobjects objectinfo("font")="Courier"
drawobjects
```

The **drawobjects** statement normally redraws objects in the current window, but it may be used with “magic windows” to redraw objects in other open windows (see “[“Magic” Windows](#)” on page 456).

Dragging a Rectangle

Dragging is the standard interface technique for moving items from one place to another. A Panorama procedure can allow a user to drag a gray rectangle from one spot to another spot. When the user releases the mouse, the procedure can be programmed to move an item or to copy data to another spot or another database (drag and drop).

The key to dragging is a special statement called **draggraybox**. This statement is designed to be used in a procedure that is triggered by a transparent button with the click/release option turned off. When the user presses on the button, the procedure is triggered immediately. The procedure calculates size and location of the original rectangle to drag around, as well as the limits to where this rectangle can be dragged. Then the **draggraybox** statement takes over. As long as the user continues to hold down the mouse a gray box will follow the mouse around. (Note: Mac OS X only allows you to drag within the current window.) When the user lets up on the mouse button the **draggraybox** statement tells the procedure the final position of the box. The procedure can then take whatever action is appropriate (moving a graphic object, copying data, etc.)

The **draggraybox** statement has four parameters. The first three of these parameters are rectangles, the fourth is a number.

```
draggraybox dragrectangle,limits,slop,axis
```

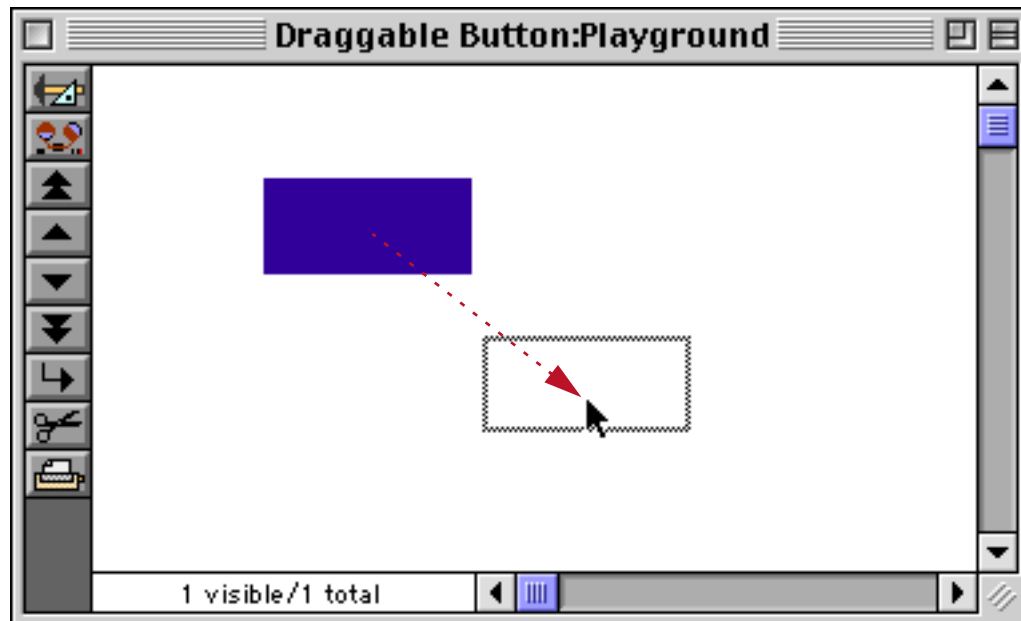
The **dragrectangle** parameter is the original co-ordinates of the rectangle the user will drag around. Often these co-ordinates are the same as the co-ordinates for the button the user pressed on. (Note: the co-ordinates for this rectangle, along with the next two, are relative to the upper left hand corner of the screen.) This parameter should be a field or variable (not a more complex formula) because after the user has released the mouse Panorama will copy the final co-ordinates into this parameter.

The **limits** parameter is the co-ordinates of a boundary rectangle that defines how far the **dragrectangle** can be dragged in each direction. For example if you don’t want the user to be able to drag the box outside of the current window you should supply the co-ordinates of the current window for limits. If the limits parameter is empty (" ") there will be no limit on how far the rectangle can be dragged. (Note: When using Mac OS X you cannot drag outside of the current window, even if the limit parameter is empty.)

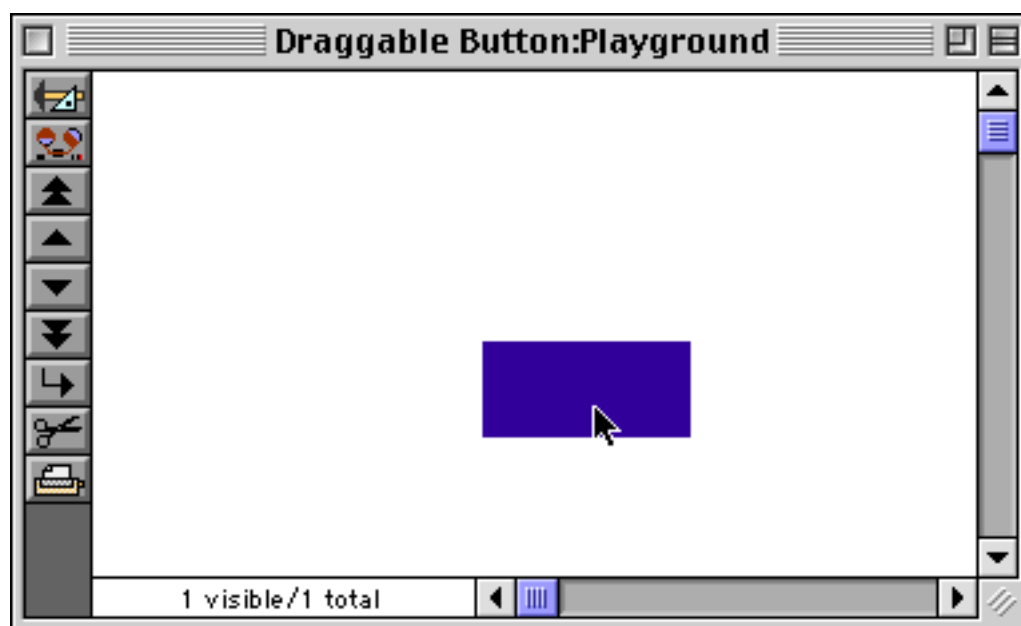
The **slop** parameter is the co-ordinates of a boundary rectangle past the limits boundary. If the user drags the mouse beyond the slop rectangle the gray rectangle will disappear completely (until the user drags back inside the slop rectangle). If the slop parameter is empty (" ") it will be the same as the limits boundary rectangle.

The **axis** parameter allows the procedure to restrict the direction the rectangle can be dragged to either horizontal or vertical. If the axis parameter is **0** the rectangle can be dragged in any direction. If the axis is **1** the rectangle can only be dragged horizontally. If the axis is **2** the rectangle can only be dragged vertically.

Here is a procedure that allows the user to drag a button around the window.



When the user releases the mouse, the procedure moves the button to the new location.



(Remember, this procedure should be triggered by a button with the click/release option turned off.)

```

1 local drag,insidewindow
2 drag=info("buttonrectangle")
3 selectobjects xytoxy(drag,"s","f")=objectinfo("rectangle")
4 insidewindow=rectangle(
  rtop(info("windowrectangle"))+20,
  rleft(info("windowrectangle"))+26,
  rbottom(info("windowrectangle"))-16,
  rright(info("windowrectangle"))-16)
5 draggraybox drag,insidewindow,info("windowrectangle"),0
6 if drag="" stop endif
7 drag=xytoxy(drag,"s","f")
8 changeobjects "rectangle",drag

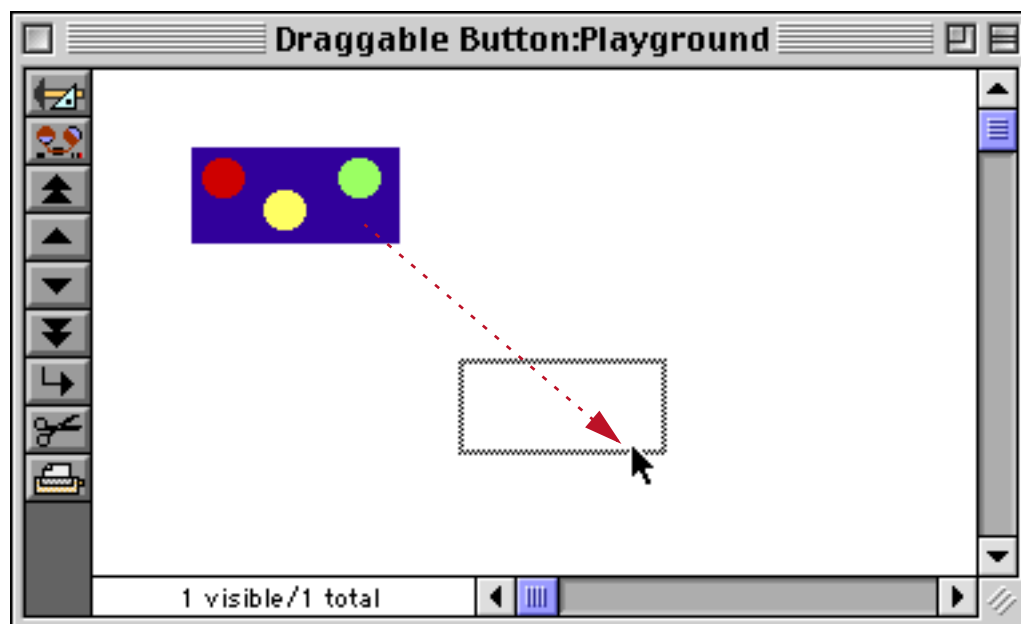
```

This example is a bit complicated, so let's take a look at it statement by statement.

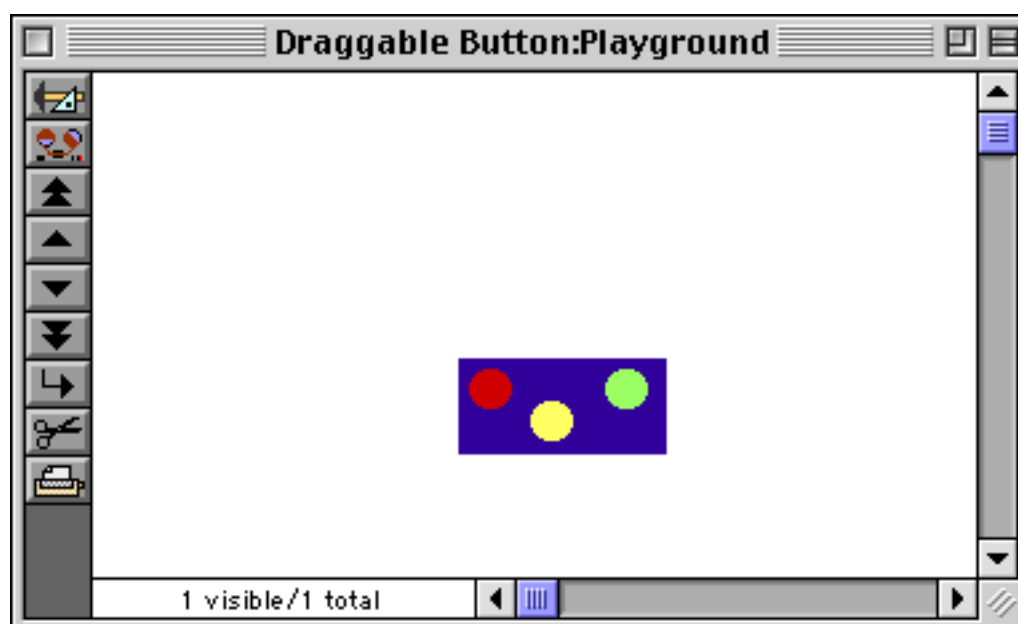
- ❶ We start by allocating the variables we need: `drag` and `insidewindow`.
- ❷ This statement finds the original location of the button, relative to the upper left hand corner of the screen.

- ③ The `selectobjects` statement selects the button the user clicked on. It identifies the button by its location on the form. If there are any other objects with the exact same dimensions, they will be selected (and moved) also.
- ④ This assignment calculates the inside dimensions of the window. It takes the raw window dimensions and moves the top down by 20 pixels (for the drag bar), the left side over by 26 pixels (for the tool palette), and the bottom and right sides in by 16 pixels (for the scroll bars). This will define the limits beyond which the button cannot be dragged.
- ⑤ Here's where dragging actually takes place. The parameters define the starting point for the drag (the original button location), the limits of dragging (the inside boundary of the window) and the limits beyond which the gray box completely disappears (the outside boundary of the window). The final parameter indicates that the button may be dragged in any direction.
- ⑥ If the user dragged the button completely out of the window the drag variable will be set to "". In that case the procedure simply stops without moving anything.
- ⑦ The new co-ordinates for the button are in `drag`. However, these co-ordinates are relative to the upper left hand corner of the screen, and the `changeobjects` statement needs them relative to the upper left hand corner of the form. The `xytoxy()` function will convert the co-ordinates.
- ⑧ The `changeobjects` statement moves the button (and any other objects with the same co-ordinates to the new position.

With a few changes the procedure can be modified to move multiple objects at once.



With this new procedure when the mouse is released all of the objects inside the boundaries of the button will move also.



This procedure moves all the objects inside the boundaries of the button.

```

❶ local drag,dragstart,insidewindow,deltaV,deltaH
❷ drag=info("buttonrectangle")
❸ dragstart=drag
❹ selectobjects
  unionrectangle(xytoxy(drag,"s","f"),objectinfo("rectangle"))
  =xytoxy(drag,"s","f")
❺ insidewindow=rectangle(
  rtop(info("windowrectangle")+20,
  rleft(info("windowrectangle")+26,
  rbottom(info("windowrectangle))-16,
  rright(info("windowrectangle))-16)
❻ draggraybox drag,insidewindow,info("windowrectangle"),0
❼ if drag="" stop endif
❽ deltaV=rtop(drag)-rtop(dragstart)
❾ deltaH=rleft(drag)-rleft(dragstart)
❿ changeobjects "rectangle",rectangle(
  rtop(objectinfo("rectangle")+deltaV,
  rleft(objectinfo("rectangle")+deltaH,
  rbottom(objectinfo("rectangle")+deltaV,
  rright(objectinfo("rectangle")+deltaH)

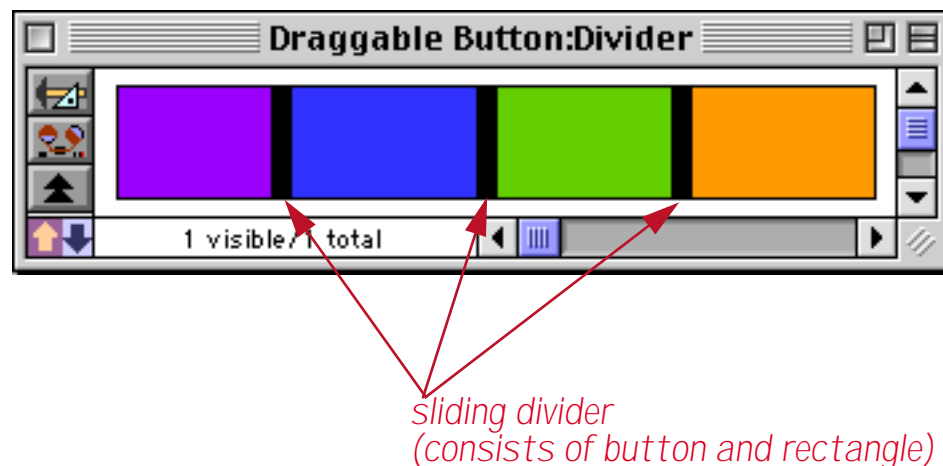
```

This procedure is similar to the last example, but with a couple of twists. Statement ❹, `selectobjects`, uses a trick with the `unionrectangle()` function (see “[Rectangles](#)” on page 149) to select all the objects inside the button. If the union of the button rectangle and object X’s rectangle is equal to the button’s rectangle then object X is completely inside the button rectangle.

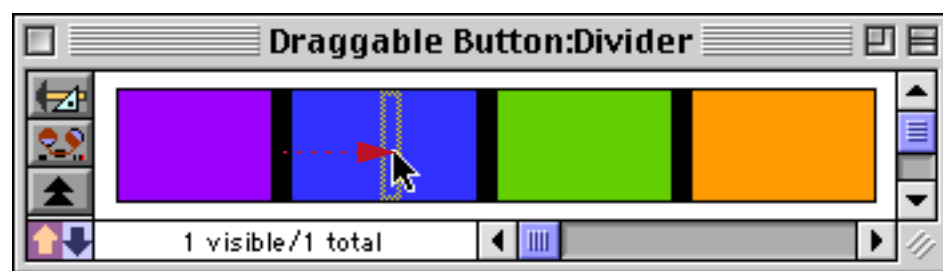
The other twist is in how the objects are moved after the drag is completed. The procedure can’t simply change all the objects to the new drag rectangle, because each object has a different position within the button. Instead, the procedure calculates the vertical and horizontal offsets between the old position and the new position (statements ❸ and ❹) and then adds this offset to each of the selected objects (statement ❿).

Movable Dividers

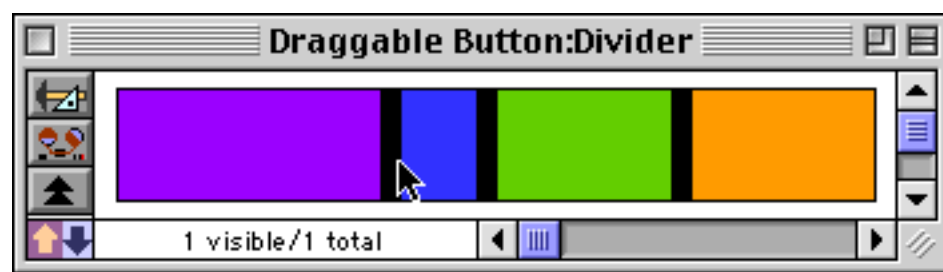
Using the `draggraybox` statement you can create a movable divider between two elements on a form. The user can slide this divider to change the division point between the two form elements. To illustrate this consider the form shown below. The form has three sliding dividers that divide the four different sections.



Each sliding divider consists of a button and a rectangle. When you press on the button a procedure is triggered (see below). That procedure allows the divider to move left or right. For example you could slide the divider between the purple and blue sections to the right.



When the mouse is released the purple section expands and the blue section gets smaller.



The dividers can be moved at any time to adjust the form as his or her needs change.

Building a sliding divider like this requires three (optionally four) graphic objects. First are the two primary elements being divided. For our example we're assuming that these two elements are side-by-side and are the same height. Between the two main elements is a small gap. This gap should be filled with a regular pushbutton (see "[Push Buttons](#)" on page 823 of the *Panorama Handbook*). The pushbutton must exactly match the gap between the two objects, so that the edges of the pushbutton are exactly on top of the edges of the primary elements. The pushbutton must have the click/release option turned off (see "[Click/Release](#)" on page 829 of the *Panorama Handbook*). You can optionally include another graphic element (for example a black rectangle or a flash art object) with the same dimensions as the pushbutton.

Here is the procedure that allows the user to slide the divider back and forth.

```

local drag,dragstart,deltaV,deltaH,slider,slidebox
drag=info("buttonrectangle")
dragstart=drag
slider=xytoxy(drag,"s","f")
slider=rectangle(
    rtop(slider),
    rleft(slider),
    rbottom(slider)+1,
    rright(slider)+1)
selectobjects
    intersectionrectangle(xytoxy(drag,"s","f"),objectinfo("rectangle"))
    ≠rectangle(0,0,0,0)
slidebox=xytoxy(objectinfo("boundary"),"f","s")
slidebox=rectangleadjust(slidebox,0,16,0,-16)
draggraybox drag,slidebox,info("windowrectangle"),1
if drag="" stop endif
deltaV=rtop(drag)-rtop(dragstart)
deltaH=rleft(drag)-rleft(dragstart)
changeobjects "rectangle",
    adjustxy(objectinfo("rectangle"),slider,deltaV,deltaH)

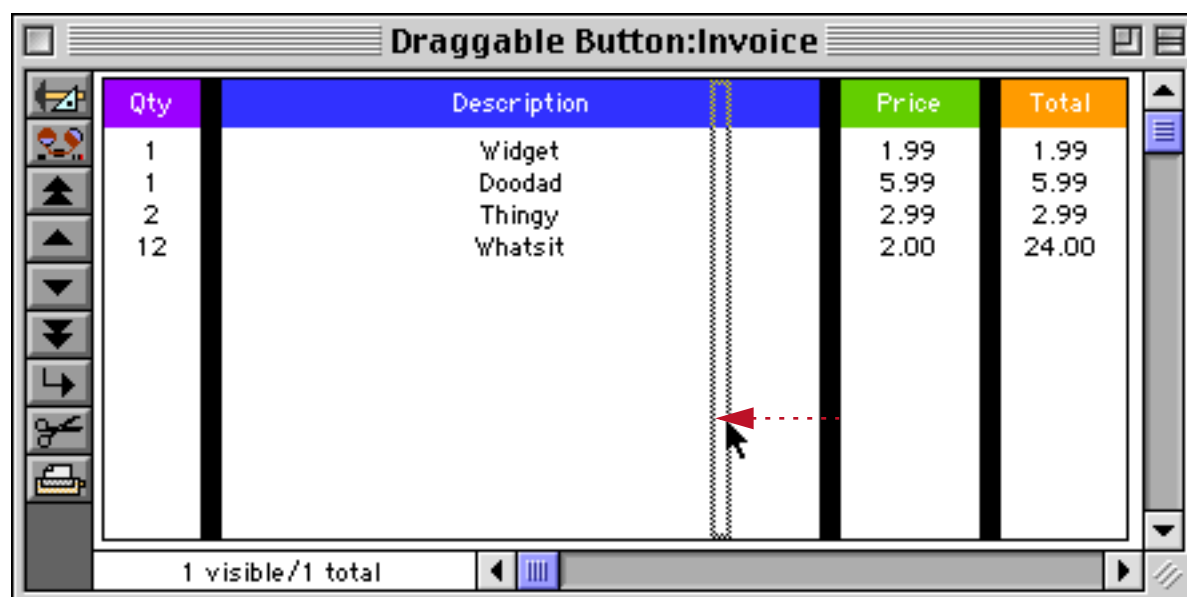
```

In this procedure, the variable `slider` is the dimensions of the button in the gap. The variable `slidebox` is the area the slider can slide back and forth in. This area includes most of the two primary elements, with a 16 pixel buffer on each end.

The last statement of this procedure uses the `adjustxy()` function to actually adjust the slider and the primary elements (see “[Rectangles](#)” on page 149). This function has four parameters: the **original rectangle**, a **boundary rectangle**, the **vertical offset** and the **horizontal offset**. The function takes the original rectangle and adjusts each corner of the rectangle by the offsets, but only if the corner is inside the boundary rectangle. If a corner is outside the boundary rectangle, it is not adjusted. Using this function it is easy for the procedure to shift the slider and gap between the two primary elements without shifting the outside edges of the primary elements.

You may have noticed that the procedure does not directly refer to either the primary objects or the slider objects. Instead it refers to everything by position. You can use this same procedure to drive several sliders in your form, or even in several forms. You can also stack several primary elements end-to-end with sliders in between each. The user can move the sliders back and forth any way they want to adjust the size of each primary element.

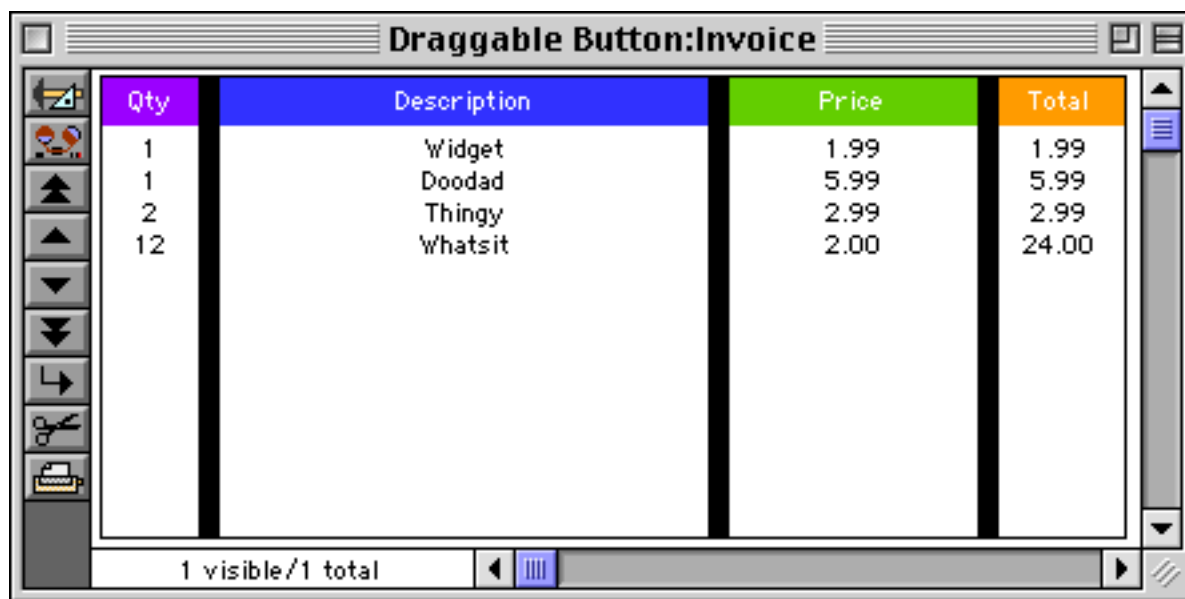
Here is an example of a more practical use for this procedure. The invoice contains four columns.



Qty	Description	Price	Total
1	Widget	1.99	1.99
1	Doodad	5.99	5.99
2	Thingy	2.99	2.99
12	Whatsit	2.00	24.00

1 visible / 1 total

The width of each column may be adjusted at any time simply by dragging on a divider (without going into graphics mode).



Qty	Description	Price	Total
1	Widget	1.99	1.99
1	Doodad	5.99	5.99
2	Thingy	2.99	2.99
12	Whatsit	2.00	24.00

1 visible / 1 total

Drag and Drop

Panorama supports the ability to drag data from one location to another (usually called drag-and-drop). On MacOS computers you can drag data within Panorama, and also drag data from other applications to Panorama or drag data from Panorama to other applications. On Windows systems you can only drag within Panorama, drag and drop between Panorama and other applications is not supported.

Drag Items and Flavors

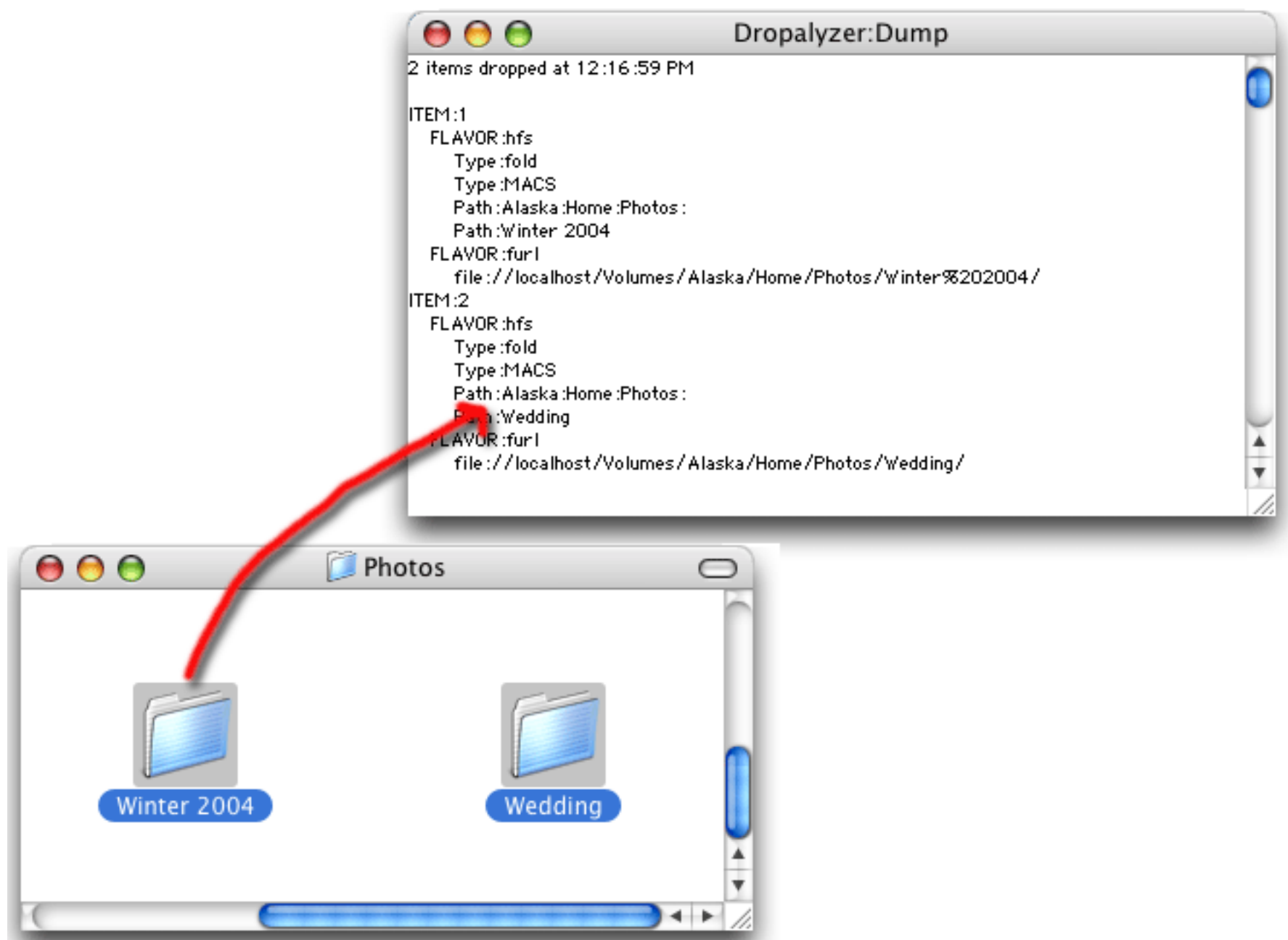
It's possible to drag all sorts of data: text, images, files, sounds, etc. It's possible to drag more than one item at a time (for example several files from the desktop). It's also possible to drag more than one type of data at a time per item, for example an image along with text describing the image. Each type of data being dragged is called a flavor. Flavors are identified by a four character code. The table below lists some of the common flavors you may encounter.

Flavor	Description
TEXT	Normal text
PICT	Picture
hfs	File or folder
furl	UNIX file path
url	Web URL
vCrd	VCard (contact info)

You can also invent your own drag flavors. Just give them a unique four letter code. Of course only your application will understand your custom flavor — you won't be able to drag items using your custom flavor to any other application. (Since you can drag more than one flavor at once you can get around this by also including one of the standard flavors, if necessary.)

The Dropalyzer Wizard

The **Dropalyzer** wizard is a handy tool for analyzing, writing and testing drag and drop procedures. You'll find this wizard in the **Developer Tools** submenu of the Wizard menu. When you first open this wizard it is completely blank, but you can drag anything you want onto this wizard and it will display some information about what was dropped. The illustration below shows the display if you drop two folders from the Finder onto the **Dropalyzer** wizard.

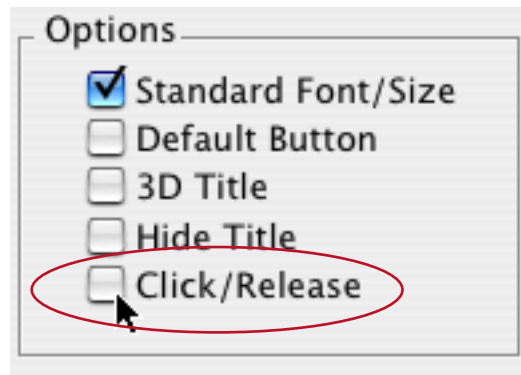


The wizard shows you that two items were dropped on it. Each item has two flavors, **hfs** and **furl**.

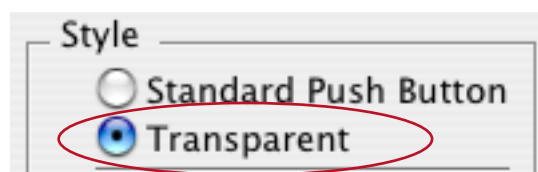
Any time you are wondering what is really going on with a drag and drop problem the Dropalyzer wizard is a lifesaver. If you are trying to figure out what an application is giving you just drop the items on the wizard. If you are trying to debug a procedure that drags from Panorama just drag to the Dropalyzer wizard and you'll see exactly what your procedure is generating.

Dragging Items from Panorama

Dragging data from Panorama always starts with clicking on a button. To allow dragging, the button's click release option must be turned off.



You can use a regular button, but usually you'll want to use a graphical button. You can either use a Flash Push Button (see "[Flash Art™ Push Button SuperObjects™](#)" on page 833 of the *Panorama Handbook*) or a regular push button with the **Transparent** option turned on. If you are using a transparent button just lay it on top of the graphic image or icon.



Like any push button you must set up a procedure that is triggered by the button. This procedure will collect the information to be dragged and start the actual drag operation. The actual code in this procedure depends on whether you are dragging one or more flavors.

Dragging a Single Flavor

If you are only dragging a single flavor the procedure only needs one statement to start the drag: **DragDrop**.

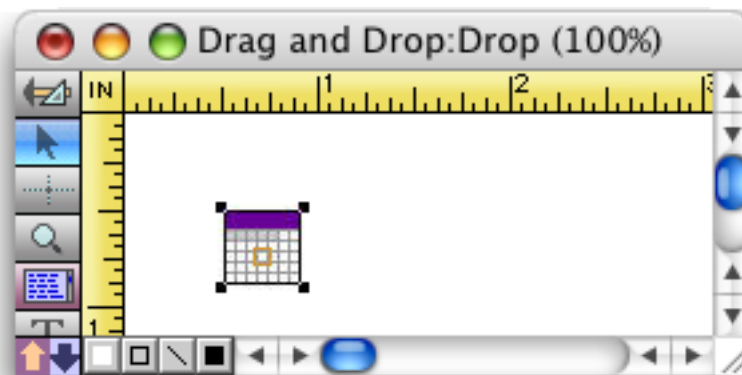
```
dragdrop rectangle,flavor,data
```

Rectangle is the dimensions of the dragged area in global co-ordinates. If you want to drag an area that is the same size as the button just use the `info("buttonrectangle")` function.

Flavor is the four letter flavor code for the data being dragged, for example "TEXT" or "PICT".

Data is the actual data to be dragged. If this is text you can simply supply a formula to calculate the text.

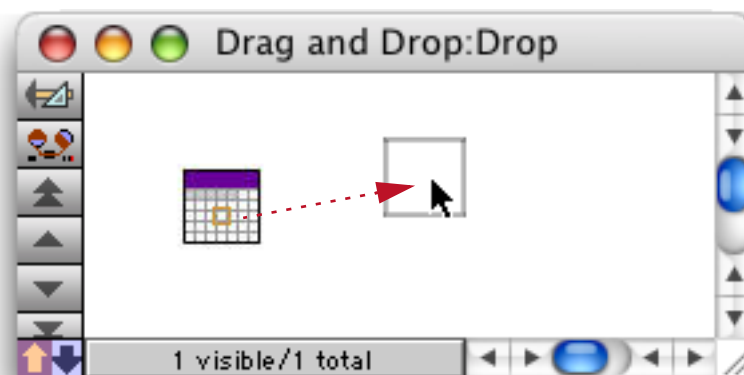
Here is an example that drags the current date. The form contains a graphic covered by a transparent push button.



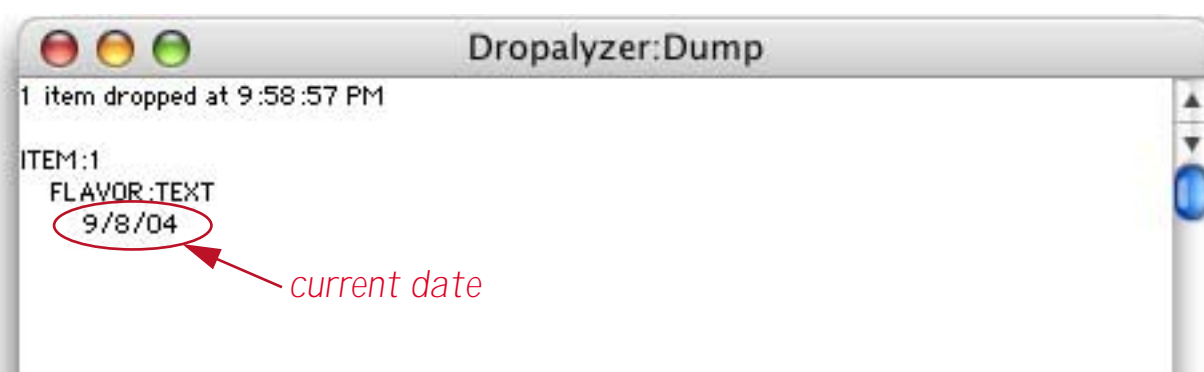
The button is designed to trigger a procedure named `.DragDate`. This procedure has a single statement:

```
DragDrop info("buttonrectangle"), "TEXT", datepattern(today(), "mm/dd/yy")
```

To copy the date to another location, click on the calendar and begin to drag.



When you drop this item on any application that accepts drag and drop text the current date will be transferred in the format `mm/dd/yy`, for example `5/30/06`.



Dragging Multiple Flavors

If you want to include more than one flavor a multi-step procedure is required. The first step is the `DragDrop` statement (see above), but the `Flavor` and `Data` parameters must be left blank (`""`). This tells Panorama that there are additional flavors to follow.

The next step is to add flavors (with data) to the drag. This is done with the `DragDropData` statement, which has two parameters: `Flavor` and `Data`. You can add as many flavors as you like by simply repeating the statement over and over.

Flavor is the four letter flavor code.

Data is the actual data to be included in the flavor.

Once all of the flavors have been added the final step is to use the `StartDragDrop` statement. This tells Panorama that all of the flavors have been specified and it should start the drag operation. This statement has no parameters.

Here is an example that drags today's date in two formats:

```
DragDrop info("buttonrectangle"), "", ""
DragDropData "TEXT", datepattern(today(), "Month ddnth, yyyy")
DragDropData "DATE", datepattern(today(), "MM/DD/YY")
StartDragDrop
```

Dragging from this button to the Dropalyzer now shows that both flavors have been included in the drag.



Note: The Dropalyzer wizard doesn't know how to display the contents of the DATE flavor, but it does show that it exists and contains 8 characters.

Receiving Dragged Data

What happens when an item is dragged and dropped onto a Panorama window? If the window is not a standard form window nothing will happen. Panorama does not support dropping on a data sheet or view-as-list window, only a standard single record form window. Any data dropped on other types of windows will be ignored.

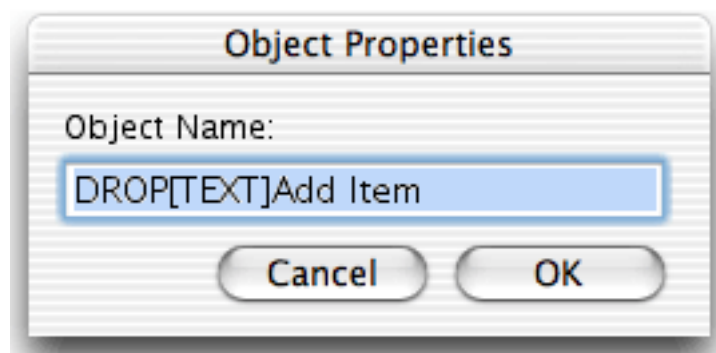
If the window dropped on is a standard form window, Panorama checks to see if the drop was on top of an object. If not, the drop is ignored.

If the data was dropped on an object, Panorama checks the name of the object. If the object name does not begin with **DROP[**, the dropped data is ignored.

If the object name contains one or more flavors followed by **]** after **DROP** (for example **DROP[TEXT]**), Panorama checks to see if data being dropped is in the list of flavors. If not, the dropped data is ignored. If the object name begins with **DROP[**, then any kind of flavor will be accepted.

If all these criteria are met then Panorama triggers the **.DropProcedure** procedure. This procedure processes the incoming dropped data, we'll discuss this procedure in more detail in a moment.

As you can see, the object name is an important component of drag and drop. Only objects with the proper name can receive dropped data. To review, you can set an object's name by selecting the object and choosing **Object Name** from the Edit menu, or by clicking on the object name in the Graphic Control Strip.



Any text in the object name after the **]** can be retrieved with the **info("dropttrigger")** function. This allows the **.DropProcedure** procedure to figure out where on the form the data was dropped. The procedure can also find out the entire object name with the **info("dropobject")** function.

The .DropProcedure

The **.DropProcedure** procedure can use several statements and functions to find out information about the drag:

info("dropttrigger") returns the trigger of the object that received the drag.

`info("dropobject")` returns the name of the object that received the drag (not just the trigger). This is handy if you use a Text Editor SuperObject as a drop object, allowing you to easily write a single procedure that allows you to drop on multiple objects.

`info("dropdatabase")` returns the name of the database that was dropped on.

`info("dropwindow")` function returns the name of the window that was dropped on. Since this may not be the current window, the first step usually taken by the `.DropProcedure` is to make this window current with this statement:

```
window info("dropwindow")
```

`info("dropform")` returns the name of the form that was dropped on

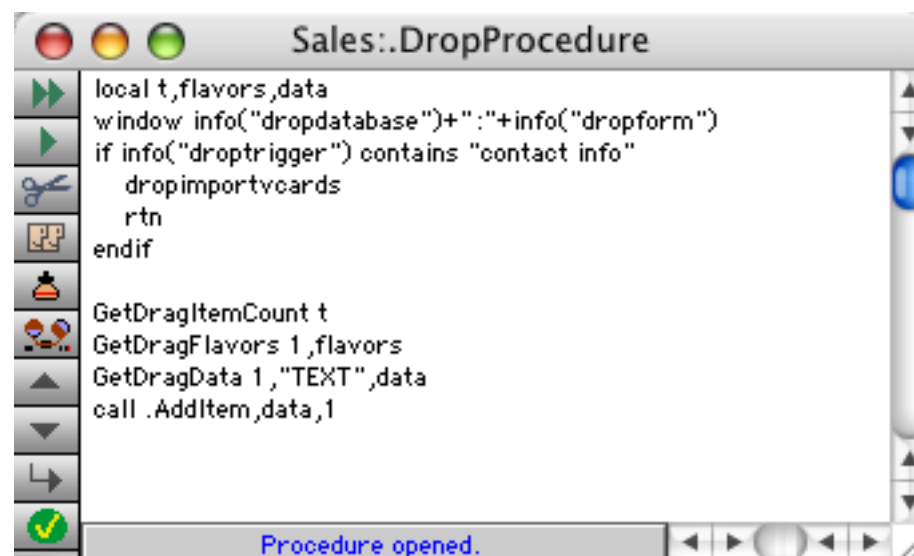
`GetDragItemCount variable` This statement returns the number of items in the drag. For example, if you select 5 items in the Finder and then drag them this statement will return 5. Items are numbered starting with 1.

`GetDragFlavors itemnum,array` This statement returns a carriage return delimited array of flavors for a particular item. Each array element is 4 characters long. The item # must be between 1 and the number of items that were dragged.

`GetDragData num,flavor,variable` This statement returns the data in a flavor.

`DragComplete` This statement frees the scratch memory used when receiving a drag. After you use this statement you can no longer use the statements and functions above (until the next drag is received).

The Sales database example included with Panorama (in the Guided Tour submenu of the Wizard menu) allows you to drag items from the Catalog to the invoice. Here is the `.DropProcedure` for this database.



The procedure starts by making sure the invoice window is the top window (the `window` statement). It then checks the drop trigger to see if the data was dropped on the contact info section of the form. If so, it uses the `dropimportvcards` statement (described later in this chapter) to import the Vcard info that was dropped (if any). Otherwise it gets the text that was dropped into a variable named `data`, and passes that to the `.AddItem` procedure. This procedure adds the item to the invoice.

Dropping Files and Folders on Panorama

Want your Panorama application to process files and/or folders that are dropped onto one of your forms? The `dropfromfinder` statement will do much of the work for you!

```
dropfromfinder filter,filelist
```

The `filter` parameter specifies types of files that you want to accept. The parameter must be 1 or more four character type codes, for example TEXT, PICT, etc. If no type codes are supplied then all files will be accepted. Any folders dropped are normally expanded (including subfolders) into a list of files contained in the folders. However, if the filter begins with `f` folders will not be expanded. In that case you would be responsible for checking for and processing folders yourself.

The `files` parameter is the name of a field or variable. When the statement is complete this field or variable will contain a carriage return separated list of every file that was dropped, including the path of the file. If any folders were dropped the statement will automatically list every file within the folder, including within subfolders (unless the `filter` parameter begins with `f`).

The example `.DropProcedure` below will accept files dropped from the finder. Any Picture files (PICT) will be added to the database in the `Photo` field.

```
local pictFiles,pf,n
dropfromfinder "PICT",pictFiles
n=1
loop
  pf=array(pictFiles,n,¶)
  stoploopif pf=""
  addrecord
  Photo=pf
  n=n+1
while forever
```

The photos added to the database could be displayed with a Flash Art SuperObject.

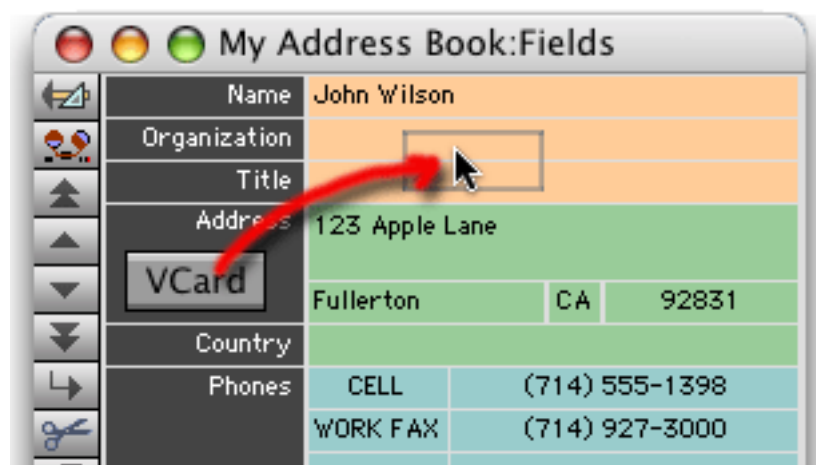
VCard Drag and Drop

Once generic fields have been set up for a database (see “[Generic Fields](#)” on page 230 of the *Panorama Handbook*) it is very easy to add drag and drop support so that you can exchange data with Apple’s Address Book (or other databases or VCard enabled applications) without using a wizard.

Dragging VCard information is just like anything else, you set up a button and a procedure. The procedure only needs a single statement:

```
dragvcard
```

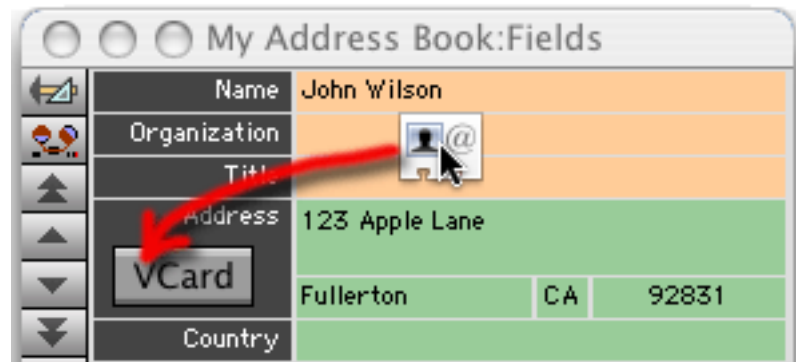
That's all there is to it! Now you can drag from your button to any VCard enabled application.



Adding the ability to drop VCards on a form is not much more difficult. The object for dropping must be set up with the proper object name, as always. This procedure only also needs only one statement

```
dropimportvcards
```

(If your database can handle other types of data dropped on it you'll need a more complex procedure that decides what has been dropped. If you detect that a VCard has been dropped, use the `dropimportvcards` statement.) Now you can drop VCards on this object to import them into the database.

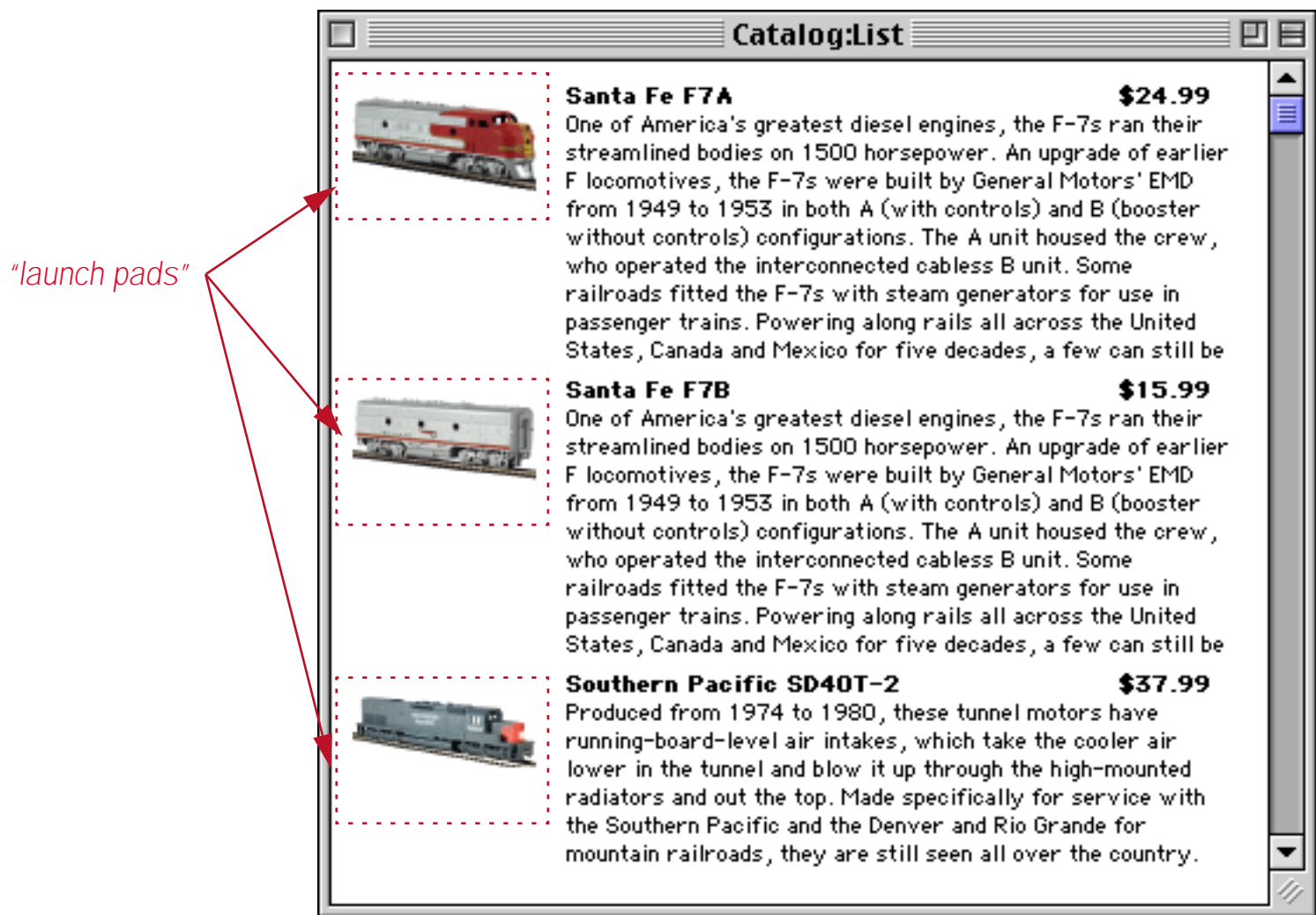


Remember, you **must** set up Generic Fields for this database before you use these statements. You'll get all kinds of error messages if you don't set Generic Fields first.

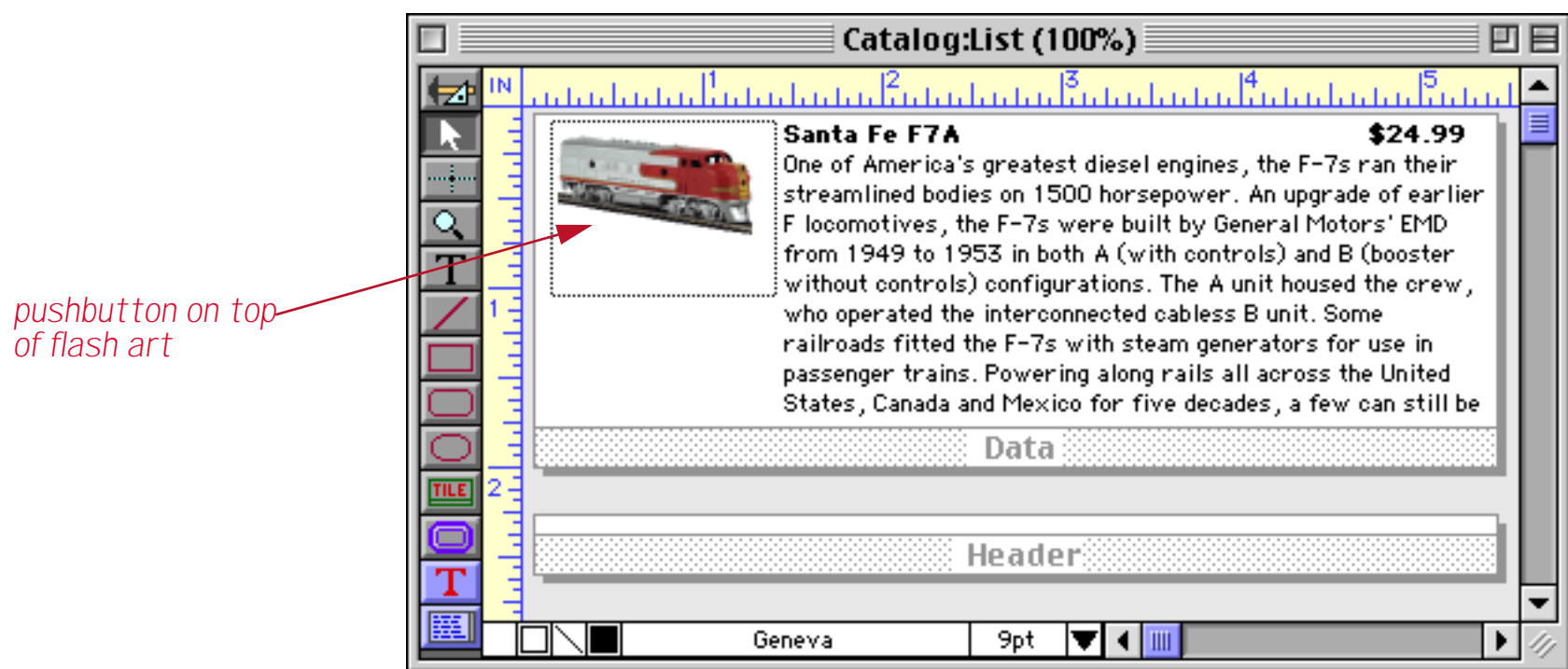
Drag and Drop (Obsolete Method)

This section describes an older technique for drag and drop that uses draggraybox. Since this statement can no longer be used to drag outside of the current window we no longer recommend this technique. We are retaining this section of the manual to help you understand any old databases that you have that may use this method. The technique will continue to work on OS 9 and Windows, but we recommend that you convert to the new method as soon as possible.

Dragging and dropping involves two active areas: a launching pad and a landing zone. Dragging starts when the user presses the mouse on an active launching pad, which is usually a pushbutton with the click/release option turned off. The user drags from the launching pad to a landing zone, an area that can receive the data from the launching pad. The landing zone may be on the same form as the launching pad, or it may be on a different form. A single launching pad can have several possible landing zones. Here is a form with three “launching pads,” one per record.



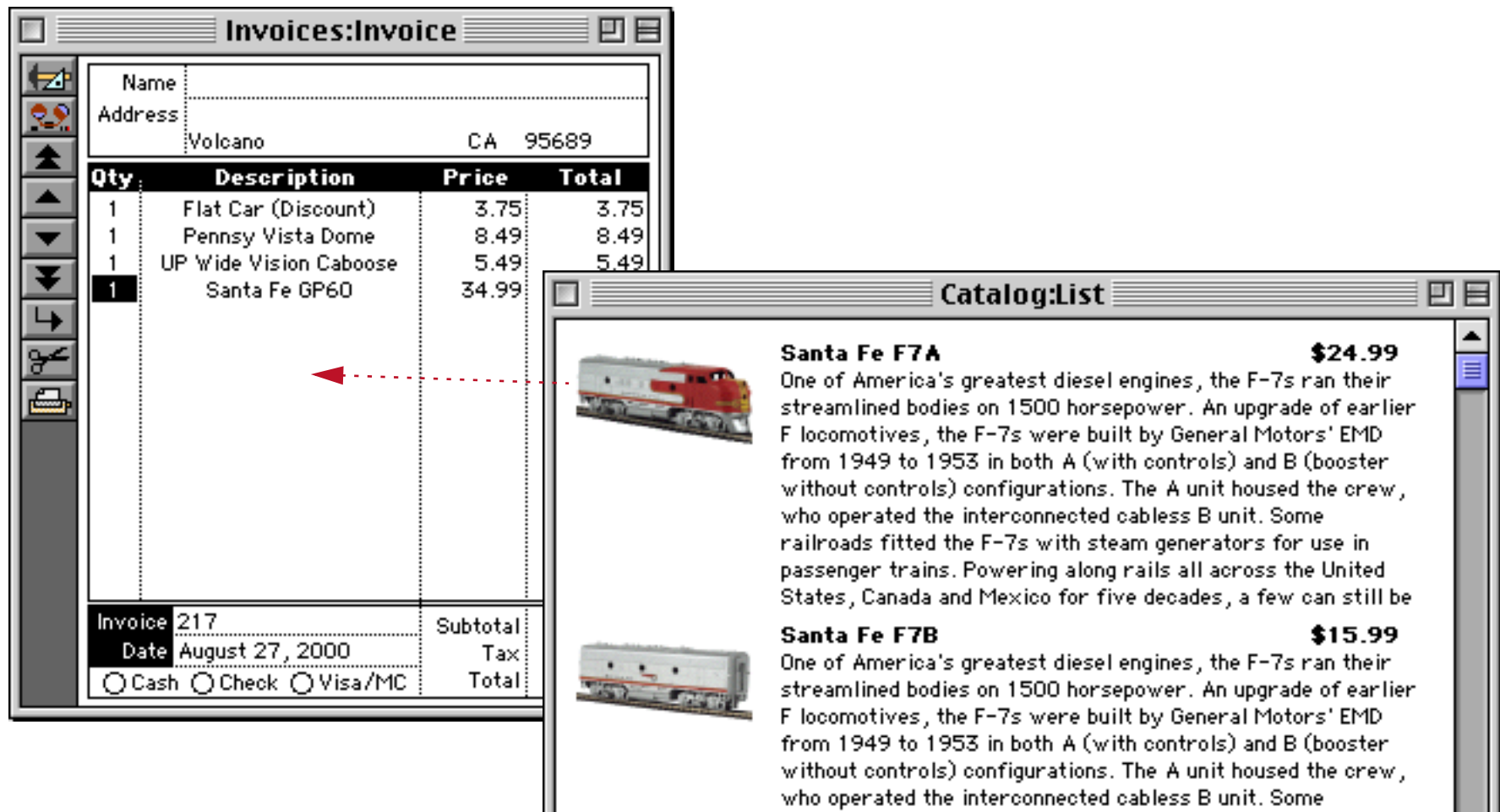
The launch pads all come from a single pushbutton on the data tile in this view-as-list form (see “[View-As-List Forms](#)” on page 899 of the *Panorama Handbook*).



The launching pad is a pushbutton. What defines a landing zone? A landing zone is whatever your procedure defines it to be. It could be an object, a collection of objects, or even an entire window.

The launch pad pushbutton triggers a procedure. This procedure uses the `draggraybox` statement to let the user drag to another location. When the user releases the mouse, the procedure must decide whether or not the mouse is over a suitable landing zone. The procedure can use the `findwindow()` function to find out what window the mouse is on top of (see “[FINDWINDOW\(\)](#)” on page 5250 of the *Panorama Reference*). If this is a window that can contain a landing zone the procedure can bring that window to the front and then use the `selectobjects` statement to find out if the mouse is over an object that is a suitable landing zone (this step is unnecessary if the whole window can be a landing zone). If the mouse is over a landing zone, the procedure then copies the data appropriately.

Now that you are familiar with the theory of drag and drop, let's take a look at some practical examples. We'll start with a catalog and invoice database, like the one's shown below. The goal is to be able to drag an item from the catalog onto the invoice and have that item added to the invoice, as shown in this illustration.



Here is the procedure that is triggered by the pushbutton. Remember, the pushbutton must have the **click/release** option turned off.

```

local drag,landingWindow,landingDatabase,landingFields
local dragItem
dragItem=Item /* copy the data for later */
drag=info("buttonrectangle") /* initial co-ordinates of box */

/* drag the box around */
draggraybox drag,"", "",0
if drag="" stop endif
landingWindow=findwindow(info("mouse"))

/* if we landed (mouse up)outside a window then stop */
if landingWindow="" stop endif

/* what database did we land on? */
landingDatabase=stripchar(landingWindow[1,":"],"!9;ÿ")

/* if landed in catalog then stop */
if landingDatabase=info("databasename") stop endif

/* does the database we landed on contain the right fields? */
landingFields=dbinfo("fields",landingDatabase)
if (not (landingFields contains "Description1" and landingFields contains "Price1" and
landingFields contains "Quantity1"))
message "Cannot drag item to this database" stop endif

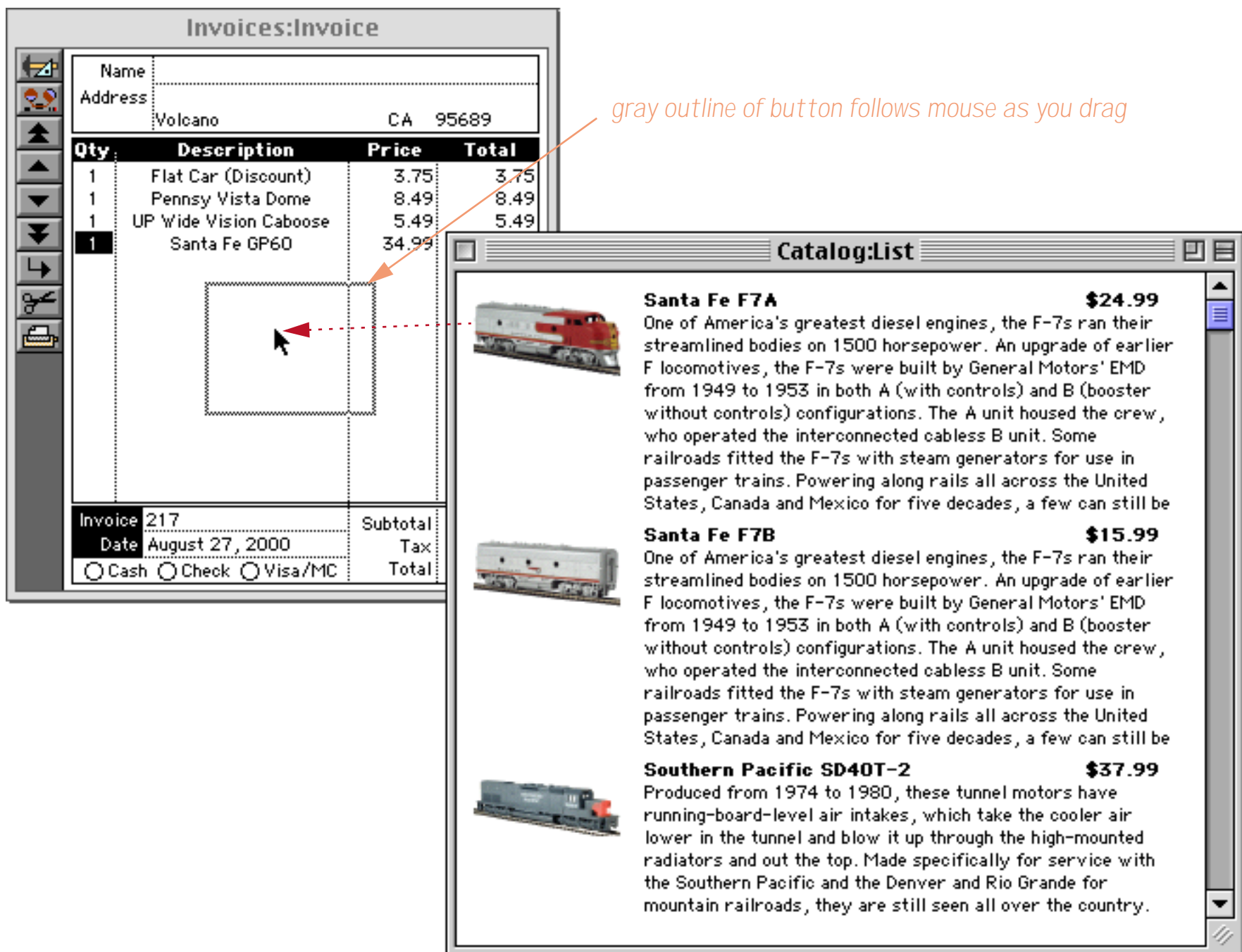
/* copy the data from the catalog into the invoice */
window landingWindow
emptyfield "QuantityΩ"
QuantityΩ=1
DescriptionΩ==dragItem

```

The procedure starts by copying the data that may be dragged into local variables (`dragItem`). Then it allows the user to drag. The drag limits are set to "", so the user can drag anywhere on the screen.

After the user releases the mouse, the procedure continues. First, it checks to see what window (if any) the user dragged to. If the user did drag to a window, the procedure strips off any extra information to figure out the name of the database. If the user released the mouse over the original database (the catalog) then the drag and drop is aborted. Otherwise, the procedure checks to see if the database has **Description**, **Price** and **Quantity** line item fields. If not, the drag and drop is aborted. If it does, the procedure brings the new window to the front and copies the data into the appropriate fields. (In this case, the landing zone is the entire window, so no further checking is required once the procedure has determined that the database the user dragged to can accept the data.

The following illustrations show the final result. When you press on one of the buttons a gray rectangle appears. This gray rectangle can be dragged over the **Invoice** database.



It doesn't matter where you release the mouse, as long as it is somewhere over the **Invoice** database. When the mouse is released the **Invoice** window comes to the front and the new item is added to the invoice. By using the double equals sign (see "[Triggering Automatic Calculations](#)" on page 523) the procedure triggers the automatic calculations built into the **Invoice** database to lookup the price and calculate the line total, subtotal, tax and grand total (as shown by the arrows).

Invoices:Invoice

Name: _____
 Address: Volcano CA 95689

Qty	Description	Price	Total
1	Flat Car (Discount)	3.75	3.75
1	Pennsy Vista Dome	8.49	8.49
1	UP Wide Vision Caboose	5.49	5.49
1	Santa Fe GP60	34.99	34.99
1	Santa Fe F7A	24.99	24.99

Invoice 217 Subtotal 77.71
 Date August 27, 2000 Tax 3.87
 Cash Check Visa/MC Total 81.58

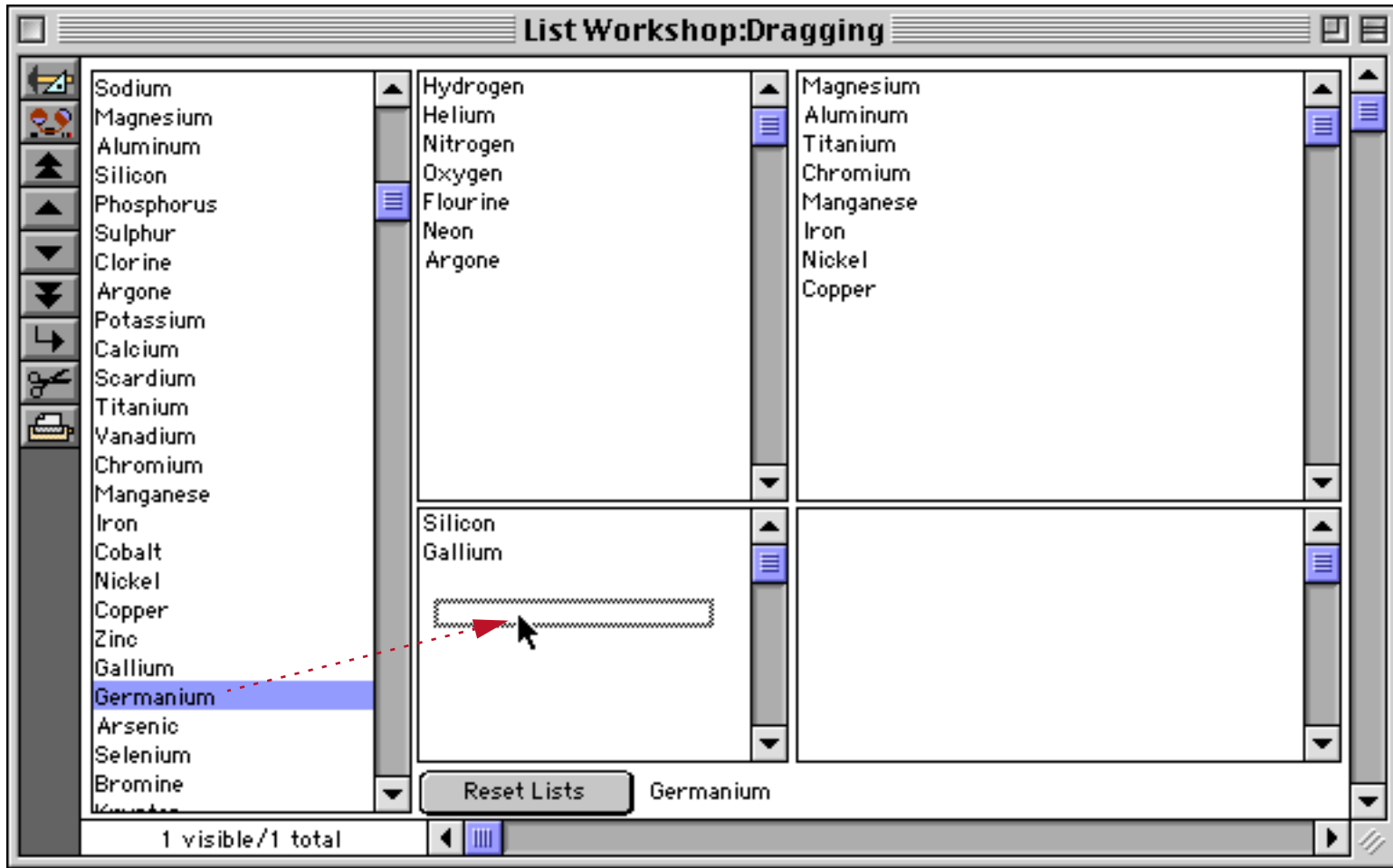
Catalog:List

Santa Fe F7A \$24.99
 One of America's greatest diesel engines, the F-7s ran their streamlined bodies on 1500 horsepower. An upgrade of earlier F locomotives, the F-7s were built by General Motors' EMD from 1949 to 1953 in both A (with controls) and B (booster without controls) configurations. The A unit housed the crew, who operated the interconnected cabless B unit. Some railroads fitted the F-7s with steam generators for use in passenger trains. Powering along rails all across the United States, Canada and Mexico for five decades, a few can still be

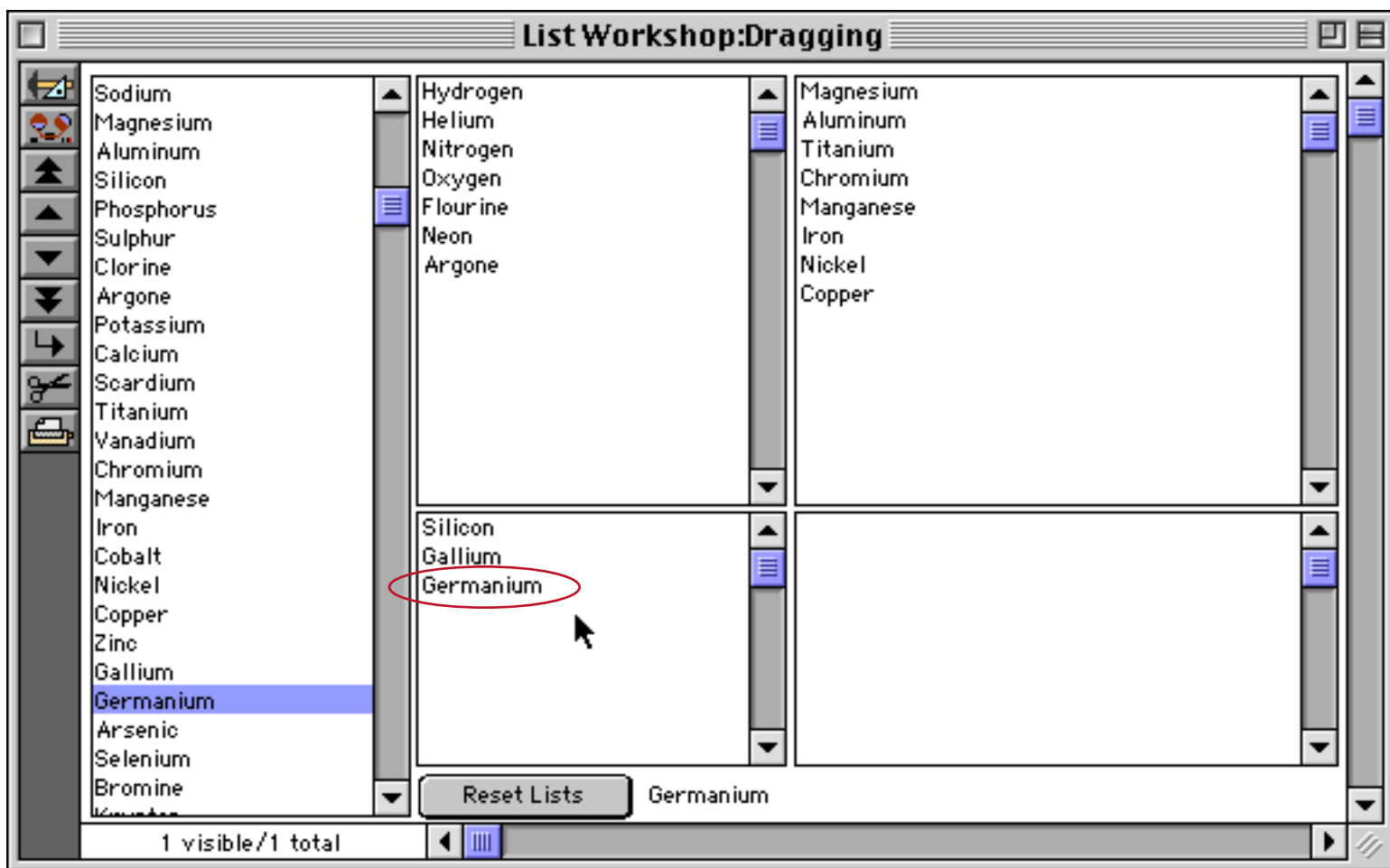
Santa Fe F7B \$15.99
 One of America's greatest diesel engines, the F-7s ran their streamlined bodies on 1500 horsepower. An upgrade of earlier F locomotives, the F-7s were built by General Motors' EMD from 1949 to 1953 in both A (with controls) and B (booster without controls) configurations. The A unit housed the crew, who operated the interconnected cabless B unit. Some railroads fitted the F-7s with steam generators for use in passenger trains. Powering along rails all across the United States, Canada and Mexico for five decades, a few can still be

Southern Pacific SD40T-2 \$37.99
 Produced from 1974 to 1980, these tunnel motors have running-board-level air intakes, which take the cooler air lower in the tunnel and blow it up through the high-mounted radiators and out the top. Made specifically for service with the Southern Pacific and the Denver and Rio Grande for mountain railroads, they are still seen all over the country.

Our second drag and drop example has one launching pad, a List SuperObject, and four landing zones, each a text display SuperObject. The finished example will allow items from the list to be dragged into one of the four landing zones.



When the mouse is released the item is dropped onto the list.



Here is the procedure that performs this drag and drop operation. It assumes that the four Text Display Objects that are acting as landing zones are named [DragList1](#), [DragList2](#), [DragList3](#) and [DragList4](#) (see "[Object Type/Object Name](#)" on page 533 of the *Panorama Handbook*) and that they are configured to display fileglobal variables with these same four names (the variables must be created in the **.Initialize** procedure).

```

/* the Click/Release option must be turned OFF!!! */

local cell,cellbox,newcell,mouse,mouseStart,landingObject,newWorkList
mouseStart=info("click")
cell=1

/* what cell did user click on */
superobject "Work List","FindCell",cell,dragItem

/* what are the dimensions of this cell */
superobject "Work List","cellrectangle",cell,cellbox

/* we need screen relative dimensions, not window relative */«
cellbox=xytoxy(cellbox,"w","s")

/* drag the box around */
draggraybox cellbox,info("windowrectangle"),info("windowrectangle"),0

/* if dragged outside of window, stop */
if cellbox="" rtn endif

/* where did we end up? */
mouse=xytoxy(info("mouse"),"s","w")

/* did we land on an object? */
selectobjects inrectangle(mouse,objectinfo("rectangle"))
objectnumber 1
landingObject=objectinfo("name")
selectnoobjects

/* if landed on one of the lists, add item to the list */
if landingObject beginswith "DragList"
  /* isn't execute cool?
     this will generate something like this:

        DragList1=sandwich("",DragList1,¶)+"Carbon" showvariables DragList1

  */
  execute landingObject+{=sandwich("",)+landingObject+{,¶)+"}+
    dragItem+" showvariables"+landingObject
endif

```


With a slight addition this procedure can also allow the main list itself to be re-arranged by dragging around the items. For example, the **Carbon** could be dragged up to the top of the list.



When the mouse is released, **Carbon** moves to the top spot and all of the other items move down.



This capability can be added by appending the steps below to the previous procedure.

```

/* if we landed on the list itself, re-arrange the order of the list */
if landingObject = "Work List"

  /* what cell did we land on? */
  superobject "Work List","pointtocell",mouse,newcell

  /* if didn't actually drag (just stayed in the same place) then stop */
  if cell=newcell stop endif

  /* check if we are off the end of the list */
  if newcell>arraysize(workList,1)
    newcell=newcell-1 /* make adjustment to stay in the list */
    newcell=0 /* add to end of list */
  endif

  /* delete dragged item from list */
  newWorkList=arraydelete(workList,cell,1,1)

  /* add dragged item back into list in the new position */
  if newcell>0
    newWorkList=arrayinsert(newWorkList,newcell,1,1)
    newWorkList=arraychange(newWorkList,dragItem,newcell,1)
  else
    newWorkList=newWorkList+1+dragItem
  endif

  /* update and display the list in the new order */
  workList=newWorkList
  superobject "Work List","FillList"
  showvariables dragItem
endif

```

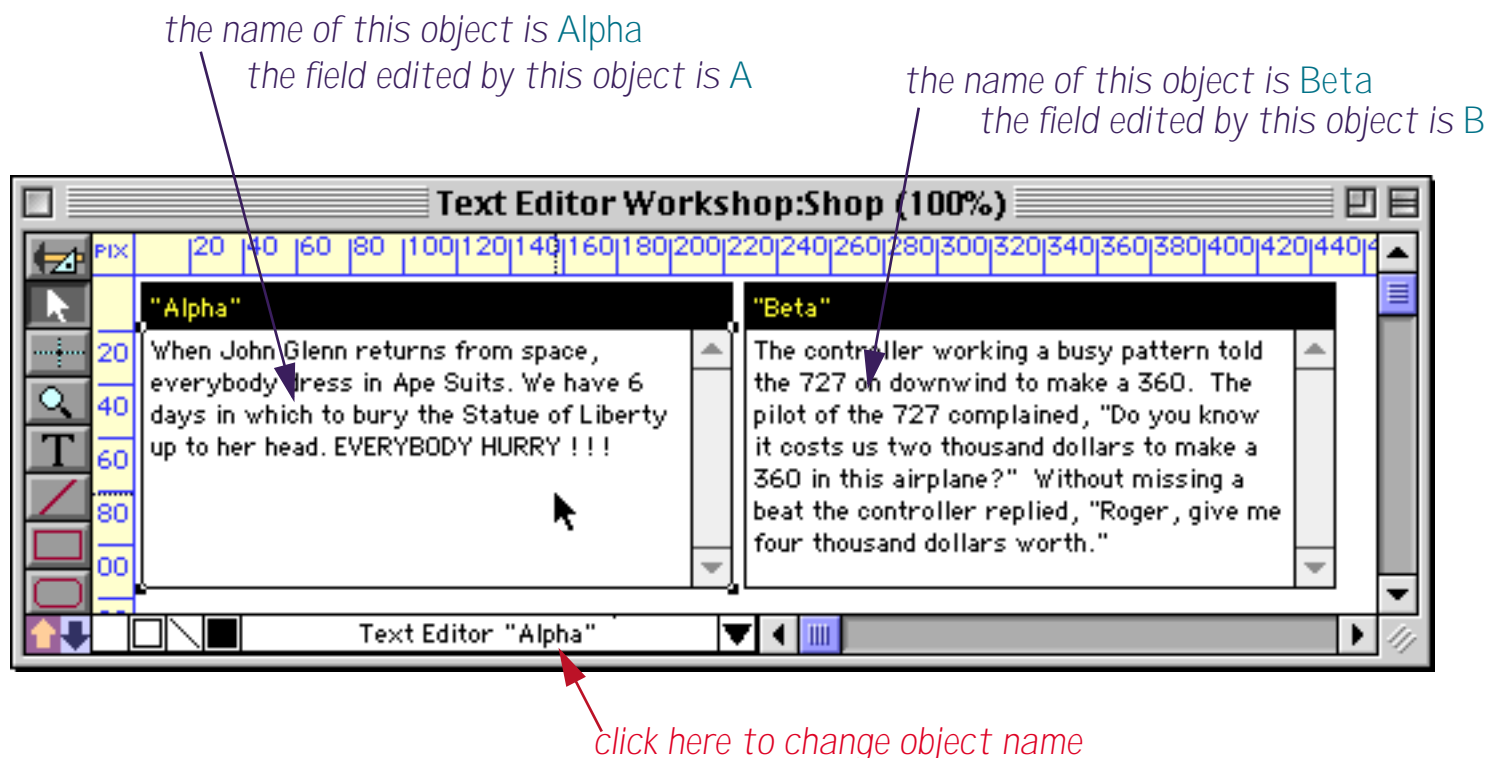
Because Panorama uses a procedure to implement drag and drop, the possibilities are endless.

Program Control of SuperObjects™

In addition to the general graphic program techniques described in the previous sections (changing position and size of objects, font, color, etc.) most types of SuperObjects™ have an additional set of specific commands that it can respond to. For example, a Text Editor SuperObject can be commanded to select a particular section of text, while a List SuperObject can be commanded to add or remove items from the list it displays. To send a command to a specific SuperObject a procedure must use the `superobject` statement.

```
superobject <name of object>,<command>,<additional parameters>
```

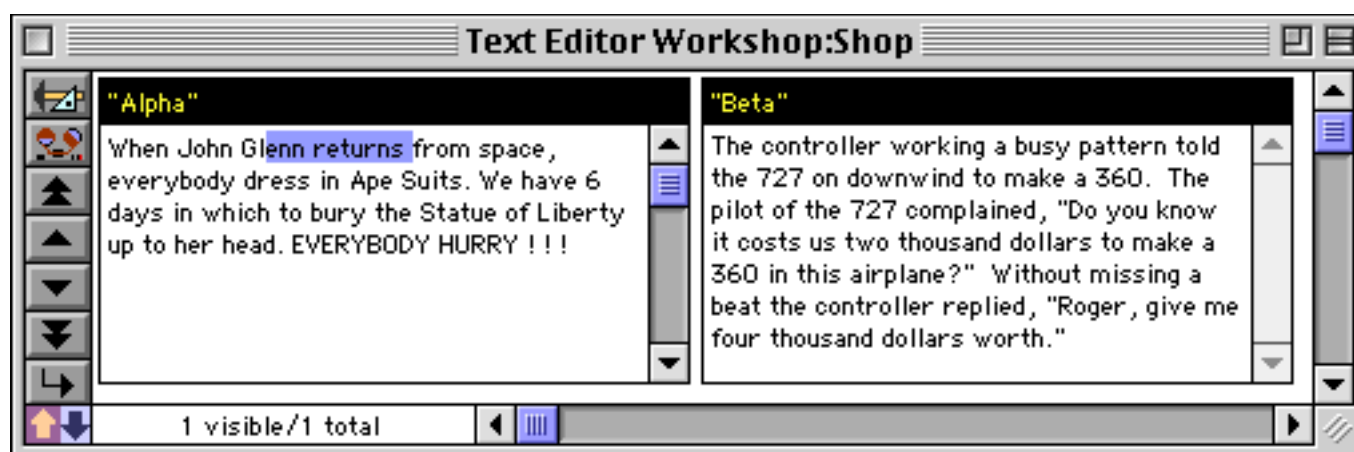
To send a command to a SuperObject the object must have a name. See “[Object Type/Object Name](#)” on page 533 of the *Panorama Handbook* to learn how to set or change the name of an object. The form shown below contains two Text Editor SuperObjects, one named **Alpha** and the other named **Beta** (these names are completely arbitrary, you can use whatever names you like). In this case the objects have been configured to edit database fields **A** and **B** respectively.



This short procedure sends two commands to the **Alpha** object (the left object). The first command tells the object to open itself for editing (the same as clicking on it). The second command tells the object to select characters 12 through 24 (the same as dragging to select these characters.)

```
superobject "Alpha","Open"  
superobject "Alpha","SetSelection",12,24
```

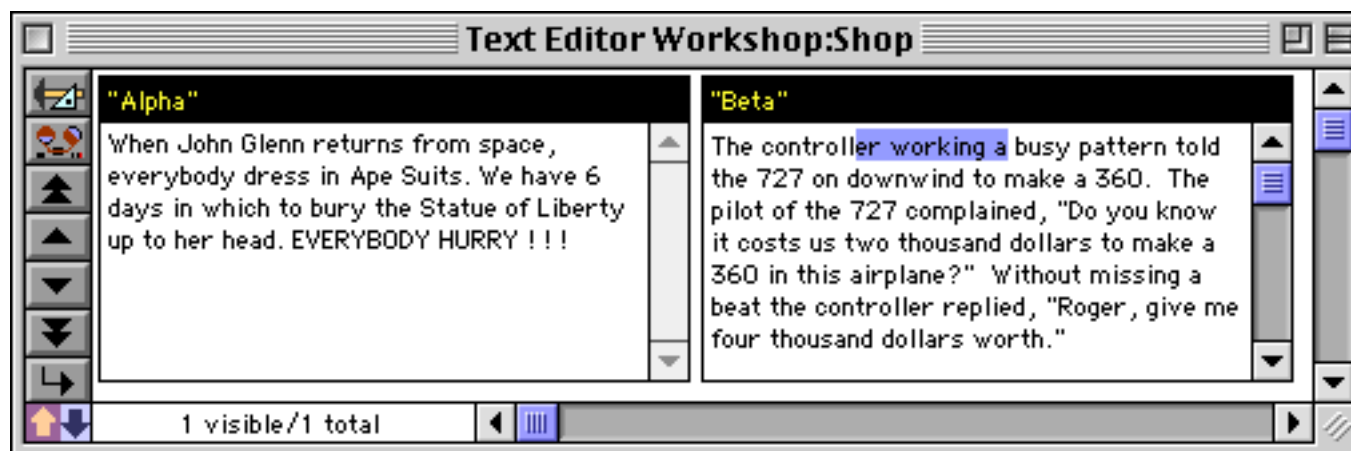
Here's the result of running this procedure.



By changing the first parameter of the `superobject` statement the procedure can control which object the commands are sent to.

```
superobject "Beta", "Open"
superobject "Beta", "SetSelection", 12, 24
```

Here's the result of running this revised procedure.



Another option is to specify the object to be manipulated in a separate `object`, `selectobjects` or `objectid` statement (see [“Selecting an Object by Name”](#) on page 633, [“Selecting Multiple Objects”](#) on page 633 and [“Object ID Values”](#) on page 643). The command will be sent to every selected object in the current form.

```
object "Beta"
superobject "", "Open"
superobject "", "SetSelection"
```

The advantage of this technique is that it makes it possible to control what objects are affected on the fly. For example you could send a command to all blue objects, or all text editor objects that appear in 12 point Times-Roman.

The `superobject` statement normally sends a command to an object(s) in the current window. If you want to send a command to an object in a different window use the `magicwindow` statement (see [““Magic” Windows”](#) on page 456).

The Active SuperObject

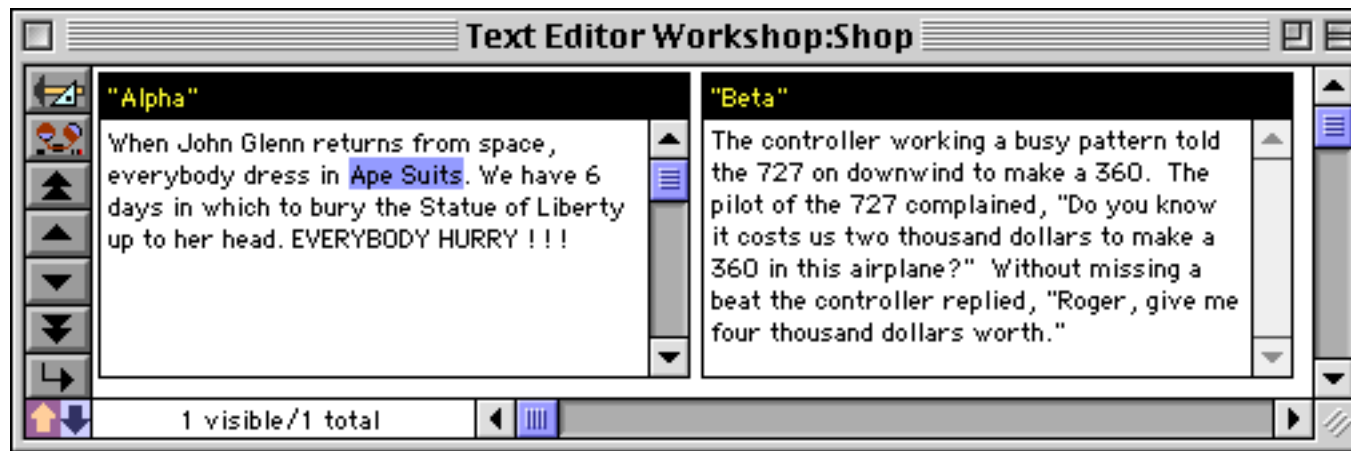
In the case of SuperObjects that edit text (Text Editor, Word Processor) only one object can be “active” at a time. The active object is the object that is currently being edited. If your procedure attempts to send a command to an editor SuperObject that is not active the procedure will stop with an error. For example the following procedure will not work.

```
superobject "Alpha", "Open"
superobject "Beta", "SetSelection", 12, 24
```

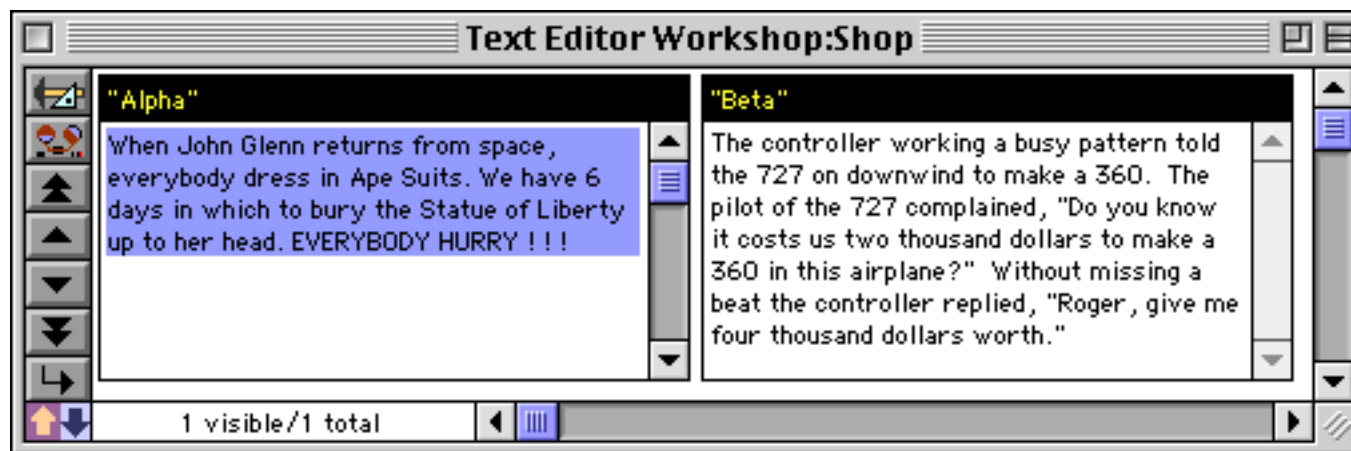
Sometimes you may want a procedure to work with whatever SuperObject happens to be open. This can be done with the `activesuperobject` statement, which always sends commands to the currently active SuperObject. Here is a procedure that will select all of the text that is currently being edited.

```
activesuperobject "SetSelection", 0, -1
```

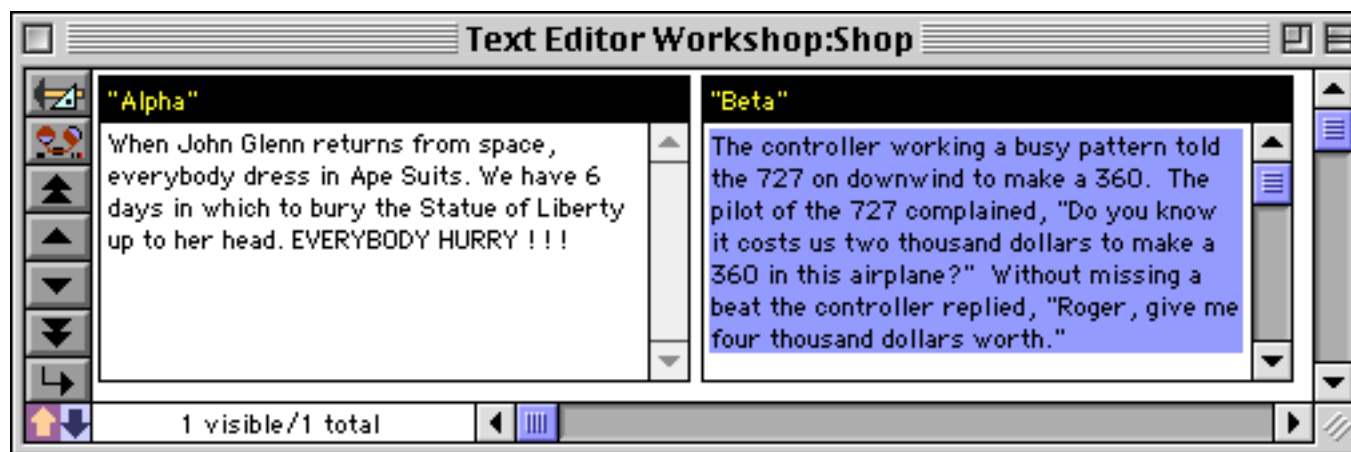
If the text on the left is being edited...



then running this procedure selects all of the text on the left.



But if the text on the right is being edited, then running this procedure selects all of the text on the right.



A procedure can find out which text editor object is active (if any) with the `info("activesuperobject")` function. This function will return the object name of the active object, or "" if no text is currently being edited.

A procedure can close the currently open text editor object with the `superobjectclose` statement. This statement checks to see if any text editing object (Text Editor or Word Processor) is currently open, and if so, closes it. If none is open, the procedure simply continues. This statement is often useful at the beginning of a procedure where you need to make sure that no text editing is happening before continuing with the procedure.

Accessing and Modifying a SuperObject's Internal Data

Most SuperObject's contain internal data for options and object status. For example a matrix object contains data that specifies the number of rows and columns, while a Super Flash Art object has internal data controlling how the image is aligned within the object. The `objectinfo()` function and `changeobjects` statement each have a “back door” that allows you to access, and in some cases modify this internal object data. Each internal data item is identified with a special identifier that may be used to access the data item. This identifier always begins with a `#` symbol, for example —

```
#SUPER MATRIX COLUMNS
#SUPER MATRIX ROWS
#SUPER FLASH ART ALIGNMENT
```

A procedure can find out what the current value of an internal data item is by using the `objectinfo()` function with the identifier for that data item. Here is an example procedure that finds out the number of rows and columns in the SuperObject Matrix named `Photo Matrix`.

```
local mCols,mRows
object "Photo Matrix"
mCols=objectinfo("#SUPER MATRIX COLUMNS")
mRows=objectinfo("#SUPER MATRIX ROWS")
```

Some (but not all) internal data items can be modified by using the `changeobjects` statement with the identifier for that data item. Here is an example that sets the Photo Matrix object to 3 rows by 4 columns.

```
selectobjects objectinfo("name")="Photo Matrix"
changeobjects "#SUPER MATRIX ROWS",4
changeobjects "#SUPER MATRIX COLUMNS",3
```

This mechanism is truly a “back door” — it changes the internal data but it does not cause the object to redraw if necessary. It's up to you as the procedure writer to force the object to redraw somehow, perhaps using the `showpage` statement or by overlaying a Text Display SuperObject with a variable so that a `showvariables` statement forces both objects to redraw.

This mechanism is a “back door” in another sense as well — it doesn't do any error checking. For some internal data items you may be able to change the value to something that doesn't make sense, does not work, or even causes a crash. Be careful, and save your work often.

Internal Data Types

Internal data items come in several flavors, as shown in this table.


Type	Description
Bit	This is a single binary digit, either 0 (off) or -1(on)
Byte	This is a number from 0 to 255 (8 bits)
Word	This is a number from 0 to 65,535 (16 bits)
Long Word	This is a number from 0 to 2,100,000,000 (32 bits)
Text	This is a string of characters

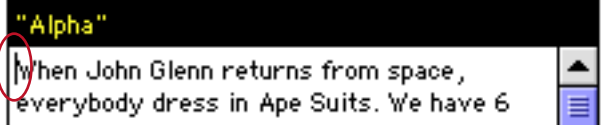
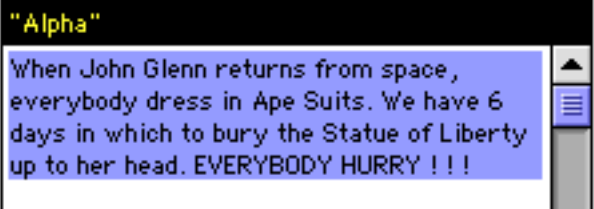
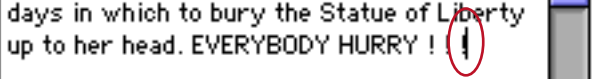
When changing an internal data type you must be careful to supply the correct type of data.

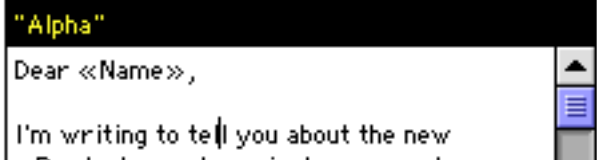
Text Editor SuperObject Commands

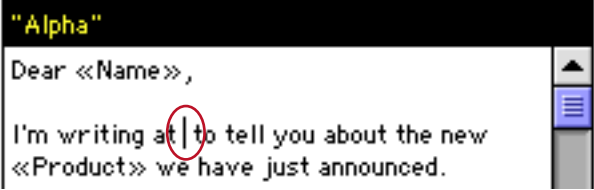
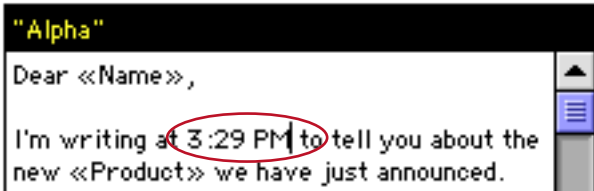
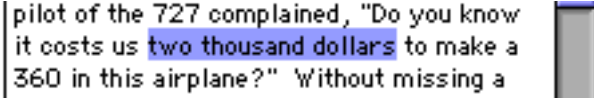
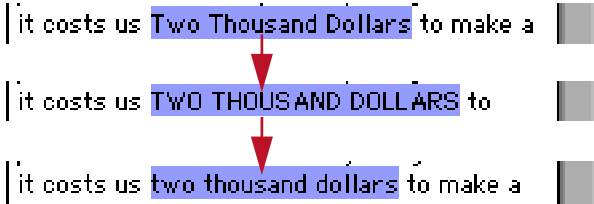
The Text Editor SuperObject understands about a dozen commands that can be sent to it with the `superobject` or `activesuperobject` statements in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). This table describes each of these commands in detail.

Command	Parameters	Description
"Open"		<p>This command opens the SuperObject for editing, if it is not already active. This command is the equivalent of clicking on the object to start editing it. Since the object isn't active yet, you can't use the <code>activesuperobject</code> statement. The example below opens the <code>Memo</code> SuperObject.</p> <pre>SuperObject "Memo", "Open" if info("ActiveSuperObject")≠"Memo" beep stop endif</pre> <p>If another data cell or SuperObject is currently active, it's possible that Panorama won't be able to open the SuperObject. If there is an error while attempting to close the currently active item (for example, incorrect date format or an illegal character in a number), the user may choose to cancel and re-edit the incorrect data. The example above checks to make sure that the SuperObject has really been opened for editing—if not, the procedure beeps and stops.</p>
"Close"		<p>This command closes the SuperObject. This is equivalent to pressing the Enter key.</p> <p>If there is an error in the data that was being edited (for example, incorrect date format or an illegal character in a number), the user may choose to cancel and re-edit the incorrect data. The procedure below checks for this and stops if this happens.</p> <pre>ActiveSuperObject "Close" if info("ActiveSuperObject")≠" " stop endif</pre> <p>An alternate method for closing the currently active SuperObject is to use the <code>SuperObjectClose</code> statement. This statement, which has no parameters, simply closes the currently active SuperObject, if any. Unlike the <code>"Close"</code> command, the <code>SuperObjectClose</code> statement will not cause an error if there is no SuperObject currently open for editing.</p>
"Cut"		<p>This command copies the currently selected text to the clipboard, then deletes the selected text. This is the same as choosing Cut from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)</p>
"Copy"		<p>This command copies the currently selected text to the clipboard, but does not delete the text. This is the same as choosing Copy from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)</p>
"Paste"		<p>This command pastes the text in the clipboard into the text being edited. The new text will replace any currently selected text, or the text will be inserted at the current insertion point if no text is currently selected. This is the same as choosing Paste from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)</p>
"Clear"		<p>This command deletes the selected text (without copying it to the clipboard). This is the same as choosing Clear from the Edit Menu. (Technical factoid: The Edit menu actually works by sending this command to the currently active SuperObject.)</p>

Command	Parameters	Description
"GetSelection"	Start,End	<p>This command gets the start and end points of the currently selected text. For the purpose of the GetSelection command (and the SetSelection command) each character is numbered, starting with zero in front of the first character. For example, if the first character was currently selected, GetSelection will return 0 and 1. If the 3rd through 8th characters are currently selected, GetSelection will return 2 and 8. If there is currently an insertion point, the starting and ending point will be the same. This command only returns the position of the selected text; if you want to get the text itself, use the "GetSelectedText" command.</p> <p>The example procedure below counts and displays the number of characters selected.</p> <pre data-bbox="971 723 1727 1020"> local SelStartPoint,SelEndPoint SelStartPoint=0 SelEndPoint=0 if info("activesuperobject")≠" ActiveSuperObject "GetSelection", SelStartPoint,SelEndPoint endif message str(SelEndPoint-SelStartPoint)+ " characters selected" </pre> <p>This procedure checks to make sure that a SuperObject is active. If there is no SuperObject active, it will display the message 0 characters selected. If the procedure did not check, Panorama would stop the procedure and display an error message if there was no active SuperObject. For example, suppose this text was selected.</p> <div data-bbox="1140 1295 1738 1453" style="border: 1px solid black; padding: 5px;"> <p>"Alpha"</p> <p>When John Glenn returns from space, everybody dress in Ape Suits. We have 6 days in which to bury the Statue of Liberty</p> </div> <p>Running this procedure will display this message.</p> <div data-bbox="1116 1592 1766 1727" style="border: 1px solid black; padding: 5px;">  28 characters selected </div>

Command	Parameters	Description
<p>"SetSelection"</p>	<p>Start,End</p>	<p>This command allows a procedure to change the selection area. It is equivalent to clicking or dragging on the text to select it. For the purpose of the "SetSelection" command (and the "GetSelection" command), each character is numbered, starting with zero in front of the first character. For example, the procedure below would put the insertion point in front of the first character in the text.</p> <pre data-bbox="971 523 1720 627"> if info("activesuperobject")≠" " ActiveSuperObject "SetSelection",0,0 endif </pre> <p>Here is the location of the insertion point after running this procedure.</p>  <p>The next example will select all of the text. Notice that the end position may be past the end of the text...Panorama will automatically adjust this for you.</p> <pre data-bbox="971 1088 1801 1193"> if info("activesuperobject")≠" " ActiveSuperObject "SetSelection",0,32768 endif </pre> <p>After this procedure is run all of the text is selected.</p>  <p>Here's a similar example that places the insertion point at the end of the text.</p> <pre data-bbox="971 1696 1878 1801"> if info("activesuperobject")≠" " ActiveSuperObject "SetSelection",32768,32768 endif </pre> <p>Here is the location of the insertion point after running this procedure.</p>  <p>This final example will increase the length of the current selection by one character.</p> <pre data-bbox="971 2163 1659 2494"> Local SelStartPoint,SelEndPoint SelStartPoint=0 SelEndPoint=0 if info("activesuperobject")≠" " ActiveSuperObject "GetSelection", SelStartPoint,SelEndPoint SelEndPoint=SelEndPoint+1 ActiveSuperObject "SetSelection", SelStartPoint,SelEndPoint endif </pre> <p>If the selection was an insertion point it will now be one character, if it was one character it will be two, if two then now three, etc.</p>

Command	Parameters	Description
"GetText"	Text	<p>This command gets all of the text being edited and puts it in a variable you specify. (Note: If you want only the selected text, use the "GetSelectedText" command.)</p> <p>The example procedure below searches for text in chevrons («») and if found, selects it. Using this procedure you could create templates with blanks to be filled in, for example ...«Gallery»...«Artist»...«Title». (Of course it might be better to store this information in fields and merge it into the text with a formula.)</p> <pre data-bbox="974 616 1902 907"> local someText,selStart,selEnd if info("activesuperobject") = "" stop endif ActiveSuperObject "GetText",someText selStart=search(someText,"«") selEnd=search(someText,"»") if selStart>0 selStart=selStart-1 endif if selEnd<selStart selEnd=selStart+1 endif ActiveSuperObject "SetSelection",selStart,selEnd </pre> <p>To illustrate this, consider the text being edited below.</p>  <p>After running the procedure, «Name» will be highlighted, like this.</p> 
"SetText"	Text	<p>This command replaces the text currently being edited with completely new text! This is a very powerful command.</p> <p>Here is a very simple example that simply erases all of the text. This is similar to Clear, except that all the text is erased, not just the selected text.</p> <pre data-bbox="974 1705 1821 1775"> if info("activesuperobject") = "" stop endif ActiveSuperObject "SetText","" </pre> <p>The next example adds a new line with a date and time stamp to the currently edited text. It also moves the insertion point to the end of the new time and date stamp, so the user can immediately type in a note.</p> <pre data-bbox="974 1968 1821 2222"> local someText if info("activesuperobject") = "" stop endif ActiveSuperObject "GetText",someText ActiveSuperObject "SetText",someText+¶+ datepattern(today(),"mm/dd/yy")+ " @"+ timepattern(now(),"hh:mm am/pm")+ " - " ActiveSuperObject "SetSelection",32768,32768 </pre> <p>Here is the result of running this procedure.</p> 

Command	Parameters	Description
"InsertText"	Text	<p>This command inserts text. The new text replaces the currently selected text, or is inserted at the insertion point if no text is selected. The example below inserts the current time into the text.</p> <pre data-bbox="971 410 1823 523">if info("activesuperobject") = "" stop endif ActiveSuperObject "InsertText", timepattern(now(),"hh:mm am/pm")</pre> <p>To use this procedure start by clicking to set the insertion point where you want the time to be inserted.</p>  <p>Running the procedure inserts the current time.</p> 
"GetSelectedText"	Text	<p>This command gets the selected text and puts it into a variable. The example below uses this command to change the case of the selected text. Each time the procedure is used the case will toggle: if the text is all lower case, it will be converted to initial caps; if it is initial caps, it will be converted to all upper case; otherwise it will be converted to all lower case.</p> <pre data-bbox="971 1456 1943 1979">local someText,editStart,editEnd if info("activesuperobject") = "" stop endif ActiveSuperObject "GetSelection",editStart,editEnd ActiveSuperObject "GetSelectedText",someText case someText=lower(someText) someText=upperword(someText) case someText=upperword(someText) someText=upper(someText) defaultcase someText=lower(someText) endcase ActiveSuperObject "InsertText",someText ActiveSuperObject "Clear" ActiveSuperObject "SetSelection",editStart,editEnd</pre> <p>To use this procedure, start by selecting some text.</p>  <p>Each time you run the procedure the text is converted to a different upper/lower case combination.</p> 

Command	Parameters	Description
"Find"		<p>This command displays a dialog asking the user what they would like to find, then locates the word or phrase within the text being edited. This is the same as using the Find in Cell command in the Edit Menu (see “Searching for Text Within the Input Box” on page 319 of the <i>Panorama Handbook</i>).</p> <p>Another way to find is to use the search() function. For an example of this, see the "GetText" command earlier in this section.</p>
"FindNext"		<p>This command locates the next occurrence of the word or phrase searched for with the "Find" command. This is the same as using the Find Next in Cell command in the Edit Menu (see “Searching for Text Within the Input Box” on page 319 of the <i>Panorama Handbook</i>).</p>
"Change"		<p>This command displays a dialog asking the user what they would like to change, then changes every occurrence it finds in the text being edited. This is the same as using the Change in Cell command in the Edit Menu (see “Replacing Words or Phrases Within a Cell” on page 321 of the <i>Panorama Handbook</i>).</p> <p>Another way to change is to use the "GetText" command and the replace() function. The example below replaces the initials rdb with Robert D. Bryce, then moves the insertion point to the end of the text.</p> <pre> local someText if info("activesuperobject") = "" stop endif ActiveSuperObject "GetText",someText ActiveSuperObject "SetText", replace(someText,"rdb","Robert D. Bryce") ActiveSuperObject "SetSelection",32768,32768 </pre>
"Spell"		<p>This command locates the next misspelled word in the text being edited. This is the same as using the Spelling command in the Edit Menu (see “Using the Spelling Checker within a Cell” on page 322 of the <i>Panorama Handbook</i>). Note: This command does not work if the optional Panorama spelling dictionary has not been installed.</p>
"GetScroll"	Vertical,Horizontal	<p>This command returns the status of the scroll bars and returns the amount scrolled (in pixels). If the text is not scrolled this will return 0,0.</p>
"SetScroll"	Vertical,Horizontal	<p>This command scrolls the text. It is the automatic equivalent to dragging on the scroll bars. To scroll to the top, use</p> <pre> activesuperobject "setscroll",0,0 </pre> <p>To scroll down four inches, use</p> <pre> activesuperobject "setscroll",72*4,0 </pre>
"GetLineCount"	Count	<p>This command queries the object to find out how many lines of text are currently being displayed. Count must be the name of a field or variable to receive the count.</p>

Text Editor Internal Data

This table describes the internal data in a Text Editor SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Text Editor Options](#)” on page 643.

Identifier	Data Type	Changeable?	Description
"#TEXT EDITOR FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — scroll bars, borders, padding, grow box, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#VERTICAL SCROLL BAR"	Bit	Yes	-1 if vertical scroll bar is enabled, 0 if disabled.
"#HORIZONTAL SCROLL BAR"	Bit	Yes	-1 if horizontal scroll bar is enabled, 0 if disabled.
"#THIN SCROLL"	Bit	Yes	-1 if thin scroll bars are enabled, 0 if disabled.
"#GROW BOX"	Bit	Yes	-1 if grow box is enabled, 0 if disabled.
"#TEXT WRAP"	Bit	Yes	-1 if wrap at end of line is enabled, 0 if disabled.
"#PROCEDURE EVERY KEYSTROKE"	Bit	Yes	-1 if procedure every key is enabled, 0 if disabled.
"#PROCEDURE MOST KEYSTROKES"	Bit	Yes	-1 if procedure most keys is enabled, 0 if disabled.
"#TOP BORDER"	Bit	Yes	-1 if top border is enabled, 0 if disabled.
"#LEFT BORDER"	Bit	Yes	-1 if left border is enabled, 0 if disabled.
"#BOTTOM BORDER"	Bit	Yes	-1 if bottom border is enabled, 0 if disabled.
"#RIGHT BORDER"	Bit	Yes	-1 if right border is enabled, 0 if disabled.
"#AQUA BORDER"	Bit	Yes	-1 if aqua borders (os x style) are enabled, 0 if disabled.
"#3D BORDER"	Bit	Yes	-1 if 3D border is enabled, 0 if disabled.
"#FOCUS RING"	Bit	Yes	-1 if blue editing focus ring is enabled, 0 if disabled.
"#TERMINATE RETURN"	Bit	Yes	-1 if terminate when return is enabled, 0 if disabled.
"#TERMINATE TAB"	Bit	Yes	-1 if terminate when tab is enabled, 0 if disabled.
"#TERMINATE UP/DOWN"	Bit	Yes	-1 if terminate when up/down arrows is enabled, 0 if disabled.
"#NON-WHITE BACKGROUND"	Bit	Yes	-1 if non-white background is enabled, 0 if disabled.
"#UPDATE VARIABLE EVERY KEY"	Bit	Yes	-1 if update variable every key is enabled, 0 if disabled.
"#FOUR SPACE TAB"	Bit	Yes	-1 if tab = 4 spaces is enabled, 0 if disabled.
"#PADDING"	Bit	Yes	-1 if padding is enabled, 0 if disabled.
"#NO NEOTEXT"	Bit	Yes	-1 if alternate editing style is enabled, 0 if disabled.
"#DROP SHADOW DEPTH"	Byte	Yes	0 if drop shadow is disabled. Non zero values specify the drop shadow offset (standard depth is 2 pixels).
"#SELECT STARTUP"	Byte	Yes	Insertion point option. 0 = end of text, 1 = beginning of text, 2 = all text selected.
"#TEXT EDITOR AUTO CAPITALIZATION"	Byte	Yes	Auto caps option. 0 = off, 1= all, 2 = word, 3 = sentence.
"#PROCEDURE"	Text	Yes	Procedure to be triggered automatically.
"#FORMULA"	Text	No	Field name, variable name, or formula.

Text Display SuperObject Internal Data

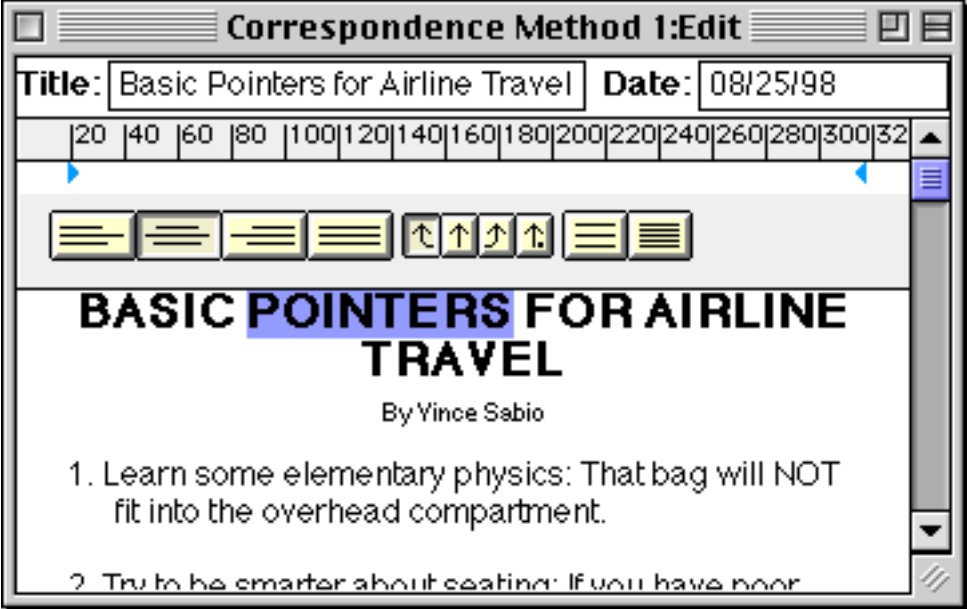

This table describes the internal data in a Text Display SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Text Display Options](#)” on page 611.

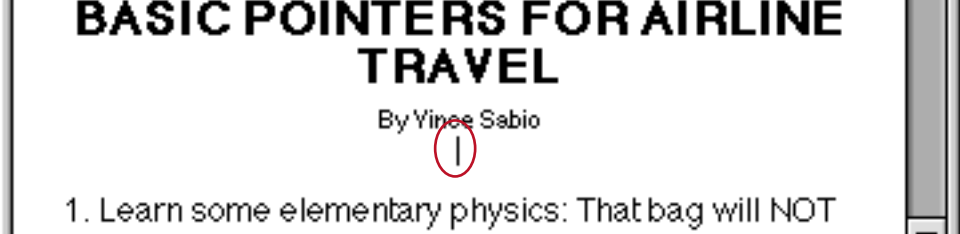
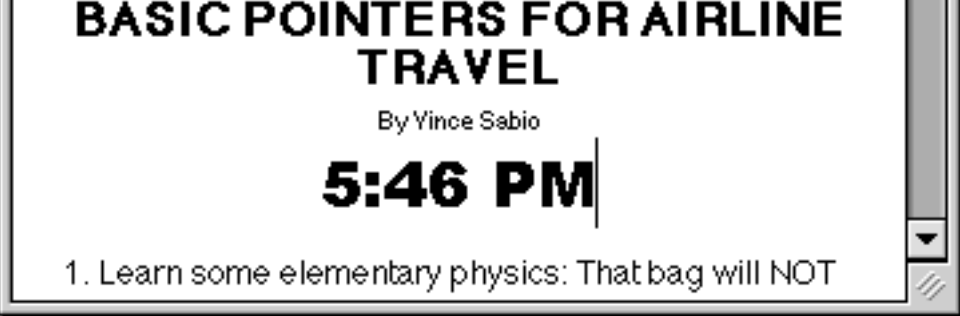
Identifier	Data Type	Changeable?	Description
"#TEXT DISPLAY FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — scroll bars, borders, alignment, grow box, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#VERTICAL SCROLL BAR"	Bit	Yes	-1 if vertical scroll bar is enabled, 0 if disabled.
"#HORIZONTAL SCROLL BAR"	Bit	Yes	-1 if horizontal scroll bar is enabled, 0 if disabled.
"#THIN SCROLL"	Bit	Yes	-1 if thin scroll bars are enabled, 0 if disabled.
"#EVALUATE FORMULA TWICE"	Bit	Yes	-1 if evaluate formula twice is enabled, 0 if disabled.
"#NO TEXT WRAP"	Bit	Yes	-1 if don't wrap text is enabled, 0 if disabled.
"#AQUA TEXT"	Bit	Yes	-1 if aqua text (anti-aliased) is enabled, 0 if disabled.
"#3D TEXT"	Bit	Yes	-1 if 3D text is enabled, 0 if disabled.
"#AUTO SIZE TEXT"	Bit	Yes	-1 if scale text size is enabled, 0 if disabled.
"#GROW BOX"	Bit	Yes	-1 if grow box is enabled, 0 if disabled.
"#TOP BORDER"	Bit	Yes	-1 if top border is enabled, 0 if disabled.
"#LEFT BORDER"	Bit	Yes	-1 if left border is enabled, 0 if disabled.
"#BOTTOM BORDER"	Bit	Yes	-1 if bottom border is enabled, 0 if disabled.
"#RIGHT BORDER"	Bit	Yes	-1 if right border is enabled, 0 if disabled.
"#DROP SHADOW DEPTH"	Byte	Yes	0 if drop shadow is disabled. Non zero values specify the drop shadow offset (standard depth is 2 pixels).
"#TEXT DISPLAY ALIGNMENT"	Byte	Yes	Align option. 0 = upper left 1 = upper center 2 = upper right 3 = middle left 4 = middle center 5 = middle right 6 = bottom left 7 = bottom center 8 = bottom right
"#TEXT DISPLAY SCALE FACTOR"	Long Word	Yes	If the #AUTO SIZE TEXT option is enabled this value controls the number of lines that will be displayed. The value is an integer that is 100 times the actual value. For example, if you want to display 2.5 lines in the text display object this value must be set to 250.
"#FORMULA"	Text	No	Formula used to display text.

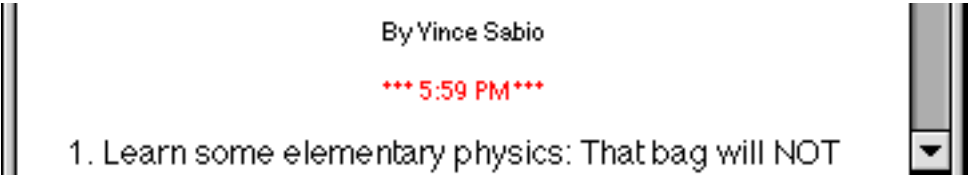
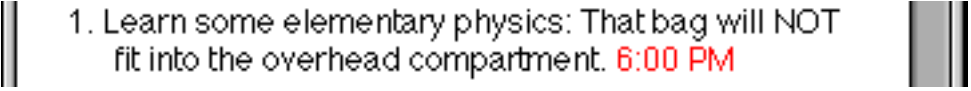
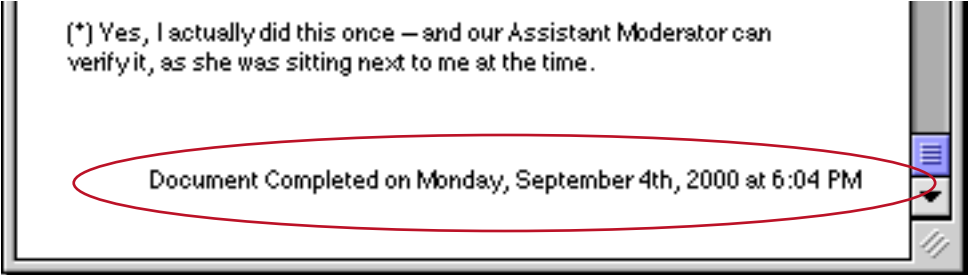
Word Processor SuperObject Commands

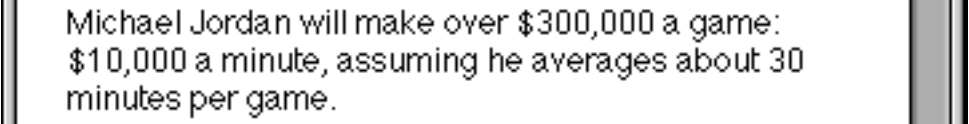
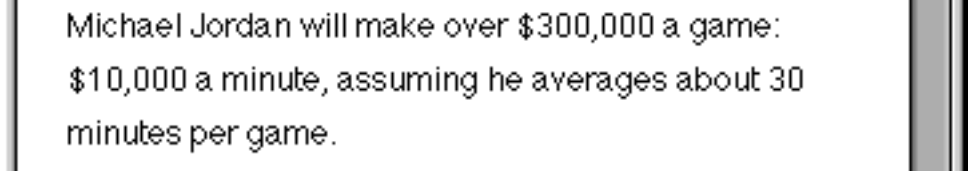
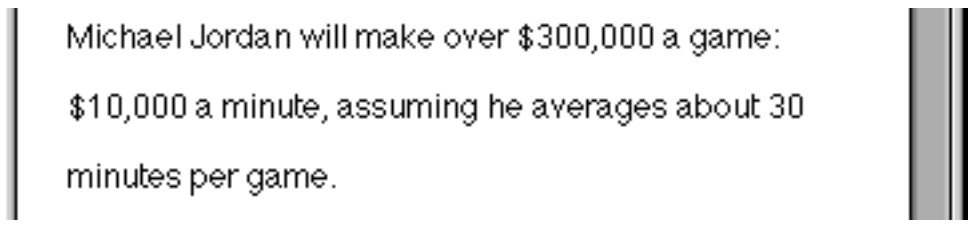
The Word Processor SuperObject understands about two dozen commands that can be sent to it with the `superobject` or `activesuperobject` statements in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). Many of these commands are identical or nearly identical to the same commands for the Text Editor SuperObject (see previous section). This table describes each of these commands in detail.

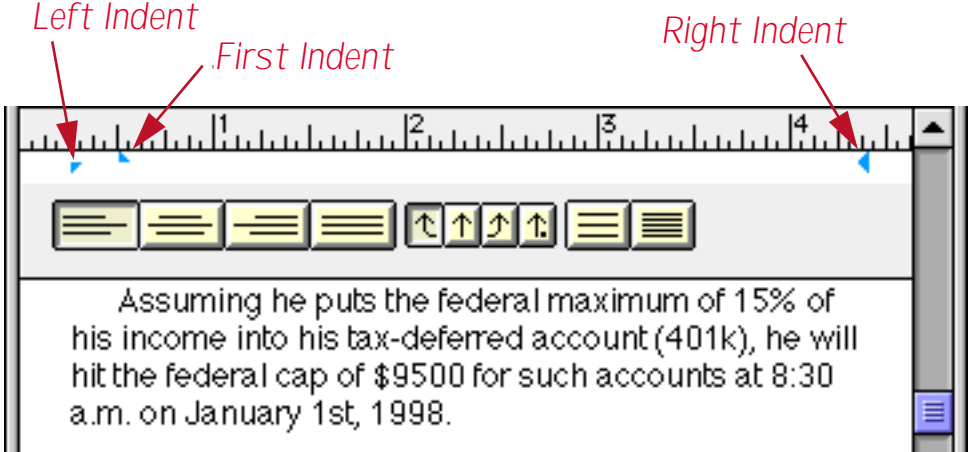
Command	Parameters	Description
"Open"		Identical to Text Editor SuperObject (see Page 670).
"Close"		Identical to Text Editor SuperObject (see Page 670).
"Undo"		Undo's the most recent editing operation.
"Cut"		Identical to Text Editor SuperObject (see Page 670).
"Copy"		Identical to Text Editor SuperObject (see Page 670).
"Paste"		Identical to Text Editor SuperObject (see Page 670).
"Clear"		Identical to Text Editor SuperObject (see Page 670).
"GetSelection"	Start,End	Identical to Text Editor SuperObject (see Page 671).
"SetSelection"	Start,End	Identical to Text Editor SuperObject (see Page 672).
"GetText"	Text	Almost identical to Text Editor SuperObject (see Page 673). However, only the text itself is copied into the variable. Style information (font, size, bold, italic, etc.) is not copied into the variable.
"SetText"	Text	Almost identical to Text Editor SuperObject (see Page 673). However, the new text is inserted into the document using the default font and style. All pre-existing style information (font, size, bold, italic, etc.) in the document is removed. To insert text without disturbing the style of existing text use the "InsertText" command.
"InsertText"	Text	Almost identical to Text Editor SuperObject (see Page 674). The new text is inserted into the document using the font and style of the text at the current insertion point.
"GetSelectedText"	Text	Almost identical to Text Editor SuperObject (see Page 674). However, only the text itself is copied into the variable. Style information (font, size, bold, italic, etc.) is not copied into the variable.
"Find"		Identical to Text Editor SuperObject (see Page 675).
"FindNext"		Identical to Text Editor SuperObject (see Page 675).
"Change"		Identical to Text Editor SuperObject (see Page 675).
"Spell"		Identical to Text Editor SuperObject (see Page 675).

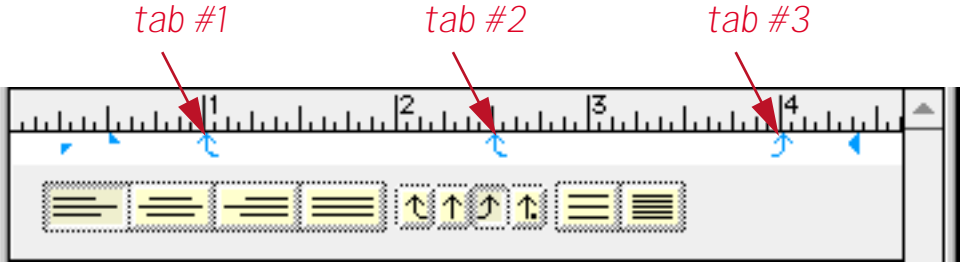

Command	Parameters	Description
"GetFont"	FontName	<p>This command gets the name of the font used for the selected text. If the selected text contains more than one font, only the first font is listed. The example below displays the font of the selected text.</p> <pre data-bbox="971 410 1779 593"> local MyFont,MyFontSize ActiveSuperObject "GetFont",MyFont ActiveSuperObject "GetFontSize",MyFontSize message "The current font is: "+ str(MyFontSize)+"pt "+MyFont </pre> <p>For example, suppose the word Pointers is selected as shown here.</p>  <p>Running this procedure displays the current font and size of this word.</p> 

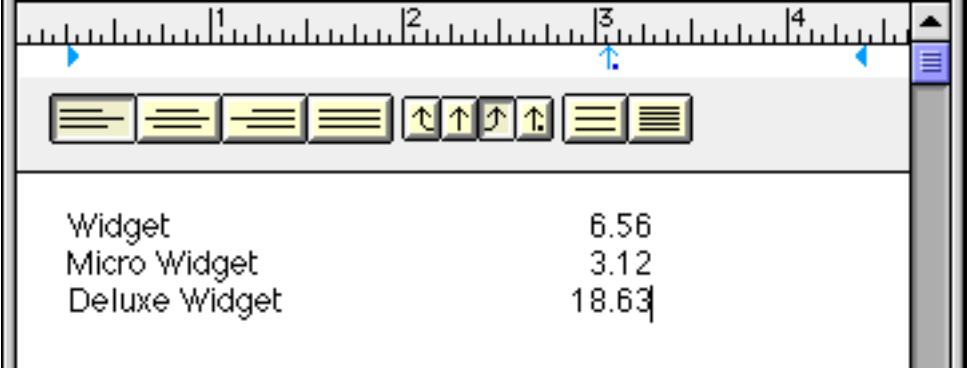
Command	Parameters	Description
"SetFont"	FontName	<p>This command changes the font of the selected text. All the selected text is changed to the same font. The example below inserts the current time into the text using 24 point Arial Black.</p> <pre data-bbox="971 410 1823 593">if info("activesuperobject") = "" stop endif ActiveSuperObject "SetFont","Arial Black" ActiveSuperObject "SetFontSize",24 ActiveSuperObject "InsertText", timepattern(now(),"hh:mm am/pm")</pre> <p>To use this procedure start by selecting an insertion point.</p>  <p>Then run the procedure to insert the time.</p>  <p>Any additional text inserted at this point will also use 24 pt Arial Black.</p>
"GetFontSize"	Size	<p>This command gets the font size of the selected text. If the selected text contains more than one size, only the first size is listed. See "GetFont" (Page 679) for an example illustrating this command.</p>
"SetFontSize"	Size	<p>This command changes the size of the selected text. All the selected text is changed to the same size. See "SetFont" (Page 680) for an example using this command.</p>


Command	Parameters	Description
"GetJustification"	Alignment	<p>This command gets the text justification status of the selected text. The result may be one of these values: Left, Center, Right or Full. If the selected text contains more than one justification, only the first justification is listed. Here is another procedure that inserts a time stamp into the file.</p> <pre data-bbox="974 483 1880 780"> local theStamp, textAlign theStamp=timepattern(now(), "hh:mm am/pm") ActiveSuperObject "GetJustification", textAlign if textAlign="Center" theStamp="*** "+theStamp+" ***" endif ActiveSuperObject "SetTextColor", rgb(65535, 0, 0) ActiveSuperObject "InsertText", theStamp </pre> <p>If the text is inserted in centered text three asterisks are added on each side of the time.</p>  <p>If the text is inserted in left or right justified text only the time is inserted.</p> 
"SetJustification"	Alignment	<p>This command changes the justification of the selected text. The new justification may be one of these values: Left, Center, Right or Full. All the selected text is changed to the same justification. The example below adds a new, right justified line to the end of the document.</p> <pre data-bbox="974 1558 1865 1889"> if info("activesuperobject") = "" stop endif ActiveSuperObject "SetSelection", 999999, 999999 ActiveSuperObject "InsertText", ¶+¶ ActiveSuperObject "SetJustification", "Right" ActiveSuperObject "InsertText", "Document Completed on "+ datepattern(today(), "DayOfWeek, Month ddnth, yyyy")+ " at "+ timepattern(now(), "hh:mm am/pm") </pre> <p>Here is the result of running this procedure.</p> 

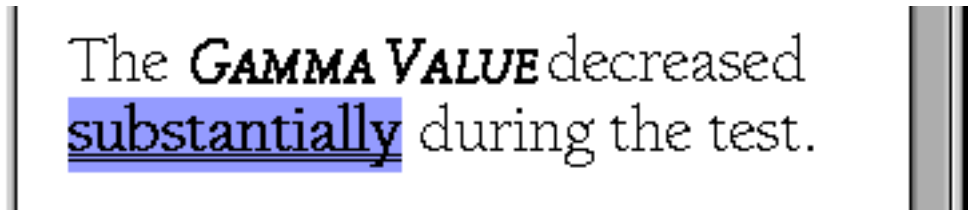
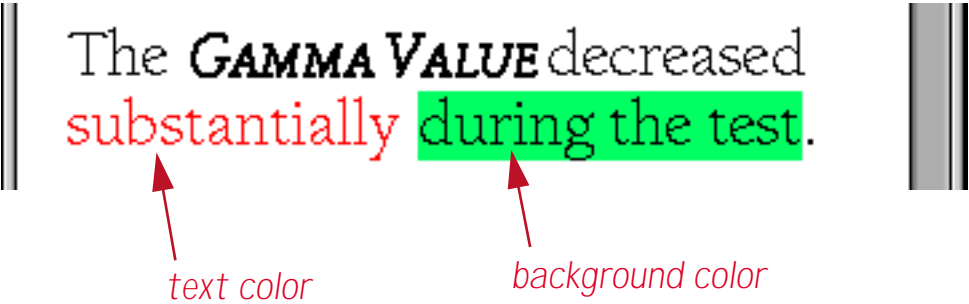
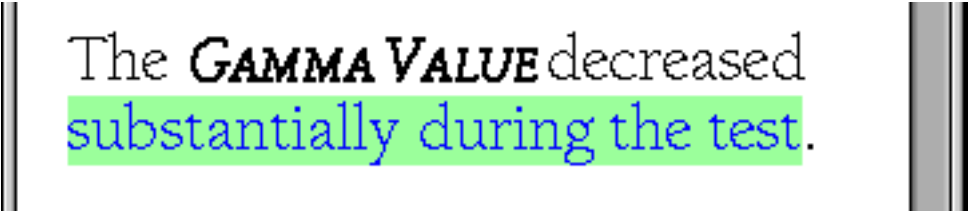
Command	Parameters	Description
"GetLeading"	Spacing	<p>This command gets the leading of the selected text (an integer). If the selected text contains more than one leading, only the first leading is returned. For normal single spaced text the leading value is zero.</p>  <p>For 12 point text a leading value of 6 is about 1 1/2 spacing.</p>  <p>This text has a leading value of 12, which is double spaced for this font size.</p>  <p>You can set the leading of selected text with the ruler or the Paragraph Settings dialog (see "Line Spacing" on page 691).</p>
"SetLeading"	Spacing	<p>This command changes the leading of the selected text. All the selected text is changed to the same leading. For normal single spaced text the leading value should be zero.</p> <p>Here is a procedure that makes the currently selected text double spaced.</p> <pre data-bbox="978 1594 1747 1702">local fontSize activesuperobject "GetFontSize",fontSize activesuperobject "SetLeading",fontSize</pre> <p>This procedure makes the currently selected text single spaced.</p> <pre data-bbox="978 1821 1589 1855">activesuperobject "SetLeading",0</pre>

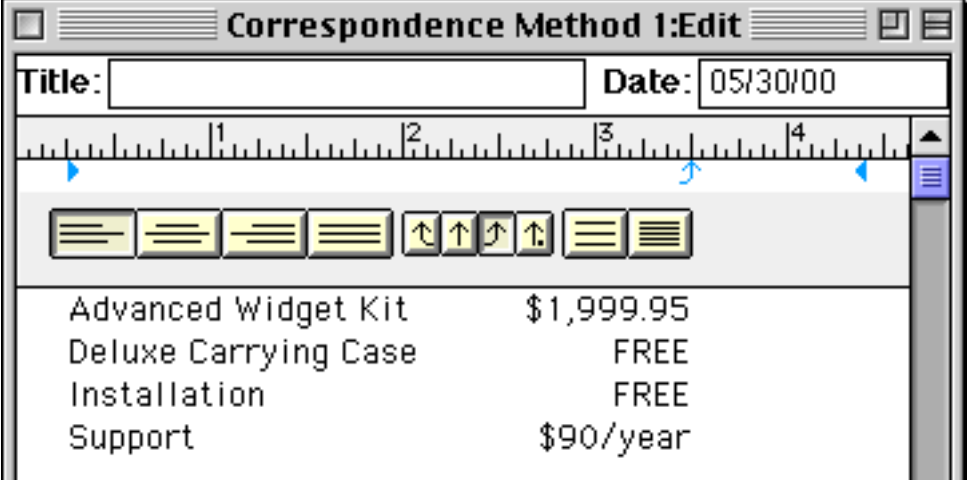
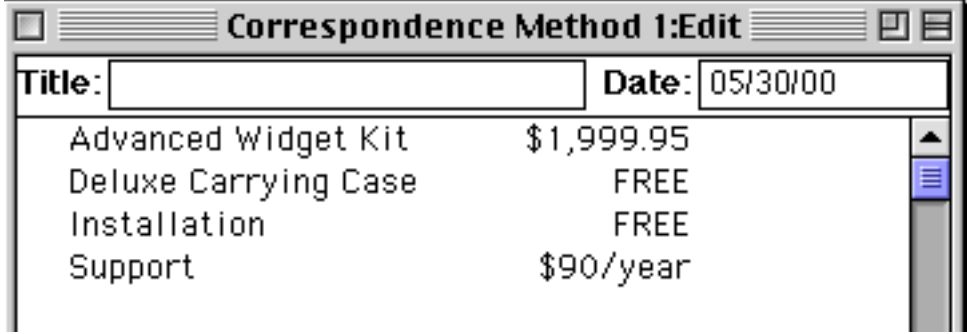
Command	Parameters	Description
"GetLeftIndent"	Indent	<p>These three commands get the indents distances of the selected text. If the selected text contains more than one indent value, only the first indent value is returned. The "GetLeftIndent" and "GetRightIndent" commands return the left and right indent values, respectively. The "GetFirstIndent" command returns the indent of the first line of the paragraph.</p>  <p>All indent values are specified in points (72 points per inch). See "Margins (Indents)" on page 682 to learn how to set indents with the ruler or Paragraph Settings dialog.</p>
"GetRightIndent"		
"GetFirstIndent"		
"SetLeftIndent"	Indent	<p>These three commands change the indents of the selected text. All the selected text is changed to the same indents. All indent values are specified in points (72 points per inch). The example below sets the margins for the currently selected text at 1/2 inch (36 points).</p> <pre>ActiveSuperObject "SetLeftIndent", 36 ActiveSuperObject "SetFirstIndent", 36 ActiveSuperObject "SetRightIndent", 36</pre>
"SetRightIndent"		
"SetFirstIndent"		
"ClearTabs"		This command clears all tabs from the selected text.

Command	Parameters	Description
<p>"GetTab"</p>	<p>Tab,Position,Type,Leader</p>	<p>This command gets information about a tab stop active with the currently selected text (see "Tab Stops" on page 685). The first parameter, Tab, is the number of the tab you want to get information about (starting with 1).</p>  <p>The remaining three parameters are filled in by the command. The Position is the position of the tab, in points. The Type is the type of tab. The possible types are Left, Center, Right, Decimal and None. A Type of None indicates that the requested tab does not exist. In that case values of the Position and Leader characters are not defined. The Leader parameter is the tab leader character, if any. The example below will display a list of the current tab stops.</p> <pre> local tabList,theTab,tabSpot,tabType,tabLeader theTab=1 tabList="" tabType="None" loop ActiveSuperObject "GetTab",theTab, tabSpot,tabType,tabLeader stoploopif tabType = "None" tabList=sandwich("",tabList,",")+ tabType+" "+pattern(tabSpot/72,"#.##")+{"} theTab=theTab+1 while forever message tabList </pre> <p>When you run this procedure it will display a message something like this.</p>  <p>There is no "SetTab" command. To change a tab setting you must clear all tabs and then use the "AddTab" command.</p>

Command	Parameters	Description						
"AddTab"	Position,Type,Leader	<p>This command adds a new tab stop. The Position is the position of the tab, in points. The Type is the type of tab. The possible types are Left, Center, Right and Decimal. The Leader parameter is the tab leader character, if any. The example below will add a tab stop and then add several lines of pricing information.</p> <pre data-bbox="971 483 1862 814"> if info("activesuperobject") = "" stop endif ActiveSuperObject "SetSelection",999999,999999 ActiveSuperObject "InsertText",¶ ActiveSuperObject "ClearTabs" ActiveSuperObject "AddTab",220,"Decimal", "" ActiveSuperObject "InsertText", "Widget"++"6.56"++¶+ "Micro Widget"++"3.12"++¶+ "Deluxe Widget"++"18.63" </pre> <p>Here is the finished result of this procedure.</p>  <table border="1" data-bbox="971 1145 1616 1266"> <tbody> <tr> <td>Widget</td> <td>6.56</td> </tr> <tr> <td>Micro Widget</td> <td>3.12</td> </tr> <tr> <td>Deluxe Widget</td> <td>18.63</td> </tr> </tbody> </table>	Widget	6.56	Micro Widget	3.12	Deluxe Widget	18.63
Widget	6.56							
Micro Widget	3.12							
Deluxe Widget	18.63							

Command	Parameters	Description																								
"GetStyle"	StyleName,Status	<p>This command will check the selected text to see if it is a certain style. If there is more than one style in the selected text, the style of the first character will be returned. If the selected text matches the specified cell the result is -1, if it does not match, the result is 0. The style names are:</p> <table border="0" data-bbox="971 438 1889 664"> <tr> <td>Plain</td> <td>Bold</td> <td>Italic</td> <td>Outline</td> </tr> <tr> <td>Shadow</td> <td>Condensed</td> <td>Extended</td> <td>Hidden Text</td> </tr> <tr> <td>Strikeout</td> <td>SuperScript</td> <td>SubScript</td> <td>SmallCaps</td> </tr> <tr> <td>AllCaps</td> <td>AllLowerCase</td> <td>FormulaMerge</td> <td>UnderLine</td> </tr> <tr> <td>DoubleUnderLine</td> <td></td> <td>WordUnderLine</td> <td></td> </tr> <tr> <td>DottedUnderLine</td> <td></td> <td>OverLine</td> <td></td> </tr> </table> <p>A text selection may contain more than one of these styles. You must test for each style separately. Here is procedure that makes a list of all the styles enabled for the first character of the currently selected text.</p> <pre data-bbox="971 848 1911 1710"> local allstyles,n,checkstyle,stylelist,styletrue allstyles="Plain,Bold,Italic,Outline,Shadow,"+ "Condensed,Extended,Hidden Text,"+ "Strikeout,SuperScript,SubScript,"+ "SmallCaps,AllCaps,AllLowerCase,"+ "FormulaMerge,UnderLine,"+ "DoubleUnderLine,WordUnderLine,"+ "DottedUnderLine,OverLine" stylelist="" n=1 loop checkstyle=array(allstyles,n,",") stoploopif checkstyle="" activesuperobject "GetStyle", checkstyle,styletrue if styletrue stylelist= sandwich(" ",stylelist,",")+checkstyle endif n=n+1 while forever message stylelist </pre> <p>This text illustrates the operation of the procedure.</p> <div data-bbox="960 1824 1932 2078" style="border: 1px solid black; padding: 10px; text-align: center;"> <p>The <i>GAMMA VALUE</i> decreased substantially during the test.</p> </div> <p>This text has three styles — bold, italic, and small caps.</p> <div data-bbox="1102 2191 1779 2361" style="border: 1px solid black; padding: 5px; text-align: center;">  <p>Bold,Italic,SmallCaps</p> </div>	Plain	Bold	Italic	Outline	Shadow	Condensed	Extended	Hidden Text	Strikeout	SuperScript	SubScript	SmallCaps	AllCaps	AllLowerCase	FormulaMerge	UnderLine	DoubleUnderLine		WordUnderLine		DottedUnderLine		OverLine	
Plain	Bold	Italic	Outline																							
Shadow	Condensed	Extended	Hidden Text																							
Strikeout	SuperScript	SubScript	SmallCaps																							
AllCaps	AllLowerCase	FormulaMerge	UnderLine																							
DoubleUnderLine		WordUnderLine																								
DottedUnderLine		OverLine																								

Command	Parameters	Description
"SetStyle"	StyleName,Status	<p>This command will change the style of the selected text. A procedure can only set one style at a time, from this list.</p> <pre>Plain Bold Italic Outline Shadow Condensed Extended Hidden Text Strikeout SuperScript SubScript SmallCaps AllCaps AllLowerCase FormulaMerge UnderLine DoubleUnderLine WordUnderLine DottedUnderLine OverLine</pre> <p>If the Status is -1, the specified style is turned on. If the Status is 0, the specified style is turned off. The style names are listed in the previous section.</p> <p>The "SetStyle" command adds or subtracts the specified style from the styles the selected text already has. If you want to make sure the selected text has only the styles you specify, start by making the text plain. The example below sets the selected text to bold double underline.</p> <pre>ActiveSuperObject "SetStyle","Plain",-1 ActiveSuperObject "SetStyle","Bold",-1 ActiveSuperObject "SetStyle","DoubleUnderLine",-1</pre> <p>Whatever text is selected when this procedure is run will be made bold with a double underline.</p> 
"GetTextColor"	Color	These two commands will return the color of the selected text. See "Colors" on page 154 for a complete discussion of colors.
"GetTextBackgroundColor"		
"SetTextColor"	Color	This command will set the color of the selected text. See "Colors" on page 154 for a complete discussion of colors. The example below sets the selected text to a pure blue on a light green background.
"SetTextBackgroundColor"		<pre>ActiveSuperObject "SetTextColor",rgb(0,0,65535) ActiveSuperObject "SetTextBackgroundColor", rgb(40000,65535,40000)</pre> <p>Here is the result of selecting text and running this procedure.</p> 

Command	Parameters	Description
"ShowRuler"	Status	<p>This command will turn the display of the ruler on and off. If the Status is -1, the ruler will be visible; if the Status is 0, the ruler will not be visible. The example below makes sure the ruler is visible.</p> <pre data-bbox="971 410 1589 446">ActiveSuperObject "ShowRuler",-1</pre> <p>The ruler allows you to manually set indents, alignment and tab stops.</p>  <p>This procedure makes the ruler invisible.</p> <pre data-bbox="971 1182 1589 1218">ActiveSuperObject "ShowRuler",0</pre> <p>With the ruler turned off you can still edit text, but changing indents, alignment and tab stops can only be done through dialogs.</p> 
"LockDocument"	Status	<p>This command allows the document to be locked. If the Status is -1, the document will be locked and cannot be edited. If the Status is 0, the document will be unlocked and may be edited again. The example below locks the current document.</p> <pre data-bbox="971 1931 1646 1968">ActiveSuperObject "LockDocument",-1</pre>

Word Processor Internal Data

This table describes the internal data in a Word Processor SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Configuring the Word Processor](#)” on page 696.

Identifier	Data Type	Changeable?	Description
"#WORD PROCESSOR FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — scroll bars, borders, padding, grow box, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#VERTICAL SCROLL BAR"	Bit	Yes	-1 if vertical scroll bar is enabled, 0 if disabled.
"#HORIZONTAL SCROLL BAR"	Bit	Yes	-1 if horizontal scroll bar is enabled, 0 if disabled.
"#TEXT WRAP"	Bit	Yes	-1 if wrap at end of line is enabled, 0 if disabled.
"#PROCEDURE EVERY KEYSTROKE"	Bit	Yes	-1 if procedure every key is enabled, 0 if disabled.
"#PROCEDURE MOST KEYSTROKES"	Bit	Yes	-1 if procedure most keys is enabled, 0 if disabled.
"#PROCEDURE ON DEACTIVE"	Bit	Yes	-1 if procedure termination is enabled, 0 if disabled.
"#TOP BORDER"	Bit	Yes	-1 if top border is enabled, 0 if disabled.
"#LEFT BORDER"	Bit	Yes	-1 if left border is enabled, 0 if disabled.
"#BOTTOM BORDER"	Bit	Yes	-1 if bottom border is enabled, 0 if disabled.
"#RIGHT BORDER"	Bit	Yes	-1 if right border is enabled, 0 if disabled.
"#TERMINATE RETURN"	Bit	Yes	-1 if terminate when return is enabled, 0 if disabled.
"#TERMINATE TAB"	Bit	Yes	-1 if terminate when tab is enabled, 0 if disabled.
"#TERMINATE UP/DOWN"	Bit	Yes	-1 if terminate when up/down arrows is enabled, 0 if disabled.
"#NON-WHITE BACKGROUND"	Bit	Yes	-1 if non-white background is enabled, 0 if disabled.
"#UPDATE VARIABLE EVERY KEY"	Bit	Yes	-1 if update variable every key is enabled, 0 if disabled.
"#3D BORDER"	Bit	Yes	-1 if 3D border is enabled, 0 if disabled.
"#GROW BOX"	Bit	Yes	-1 if grow box is enabled, 0 if disabled.
"#OVERFLOW PRINTING"	Bit	Yes	-1 if handle overflow is enabled, 0 if disabled.
"#FILE ON DISK"	Bit	Yes	-1 if file on disk is enabled, 0 if disabled.
"#DROP SHADOW DEPTH"	Byte	Yes	0 if drop shadow is disabled. Non zero values specify the drop shadow offset (standard depth is 2 pixels).
"#TEXT EDITOR AUTO CAPITALIZATION"	Byte	Yes	Auto caps option. 0 = off, 1= all, 2 = word, 3 = sentence.
"#PROCEDURE"	Text	Yes	Procedure to be triggered automatically.
"#FORMULA"	Text	No	Field name, variable name, or formula.

Super Flash Art Commands (Including Movie Control)

The Super Flash Art SuperObject understands about a dozen commands that can be sent to it with the `superobject` statement in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). Many of these commands work with QuickTime movies (see “[Displaying Movies in a Form](#)” on page 819). This table describes each of these commands in detail.

Command	Parameters	Description
"Dimensions"	Rectangle	<p>This command obtains the original height and width of the currently displayed image and places it in the rectangle you supply (usually a variable, see “Rectangles” on page 149). The dimensions are in pixels (1/72 inch). You should use the <code>rheight</code>(and <code>rwidth</code>(functions to extract the height and width of the rectangle.</p> <p>Here is a procedure that examines the photo being displayed in the object named <code>Photo</code> and decides whether it is portrait, landscape, or panoramic.</p> <pre> local photoRect,photoHeight,photoWidth superobject "Photo", "Dimensions",photoRect photoHeight=rheight(photoRect) photoWidth=rwidth(photoRect) case photoWidth>photoHeight*2 message "Panoramic Photo" case photoWidth>photoHeight message "Landscape Photo" default message "Portrait Photo" endcase </pre>
"FindText"	Point,Text	<p>This command only works when displaying an Apple PICT format image that contains vector text (not bitmap). (See “Preparing Pictures with Extractable Text” on page 697.) When the procedure issues this command it must supply an x,y position (<code>Point</code>) within the PICT image. The command will scan the image and see if there is any text at this point. If so, the command will fill in the <code>Text</code> parameter (usually a variable) with the text. This command is usually used to create a web like hypertext system within Panorama — see “Building Web Like HyperText Systems with Super Flash Art” on page 697.</p>
"ExtractText"	Font,Size,Style,Sep,Text	<p>This command only works when displaying an Apple PICT format image that contains vector text (not bitmap). (See “Preparing Pictures with Extractable Text” on page 697.) The procedure specifies what <code>Font</code>, <code>Size</code>, and <code>Style</code> it wants to extract (for example "<code>Helvetica</code>",<code>12</code>,<code>0</code>) and the command scans the image looking for text that matches. If it finds any it is placed in the <code>Text</code> parameter. If there is more than one section of text that matches the sections are appended together with the <code>Sep</code> character in between. For more detail about this command see “Extracting All Text of a Specific Style” on page 701.</p>
"GetDuration"	Time	<p>This command allows you to get the duration of a movie. The result is in 600ths of a second. This example displays the length of the currently playing movie.</p> <pre> local mTime superobject "MyMovie", "GetDuration",mTime message "Movie Length: "+str(mTime/600)+" seconds" </pre>

Command	Parameters	Description
"GetRate"	Rate	<p>These commands allows you to get and set the current movie playback speed. If a movie is stopped, the rate is zero. Normal speed is 65536, 1/2 speed (slow motion) is 32768, double speed is 131072. The example below cuts the current playback speed in half.</p> <pre>local mSpeed superobject "MyMovie", "GetRate", mSpeed superobject "MyMovie", "SetRate", mSpeed/2</pre>
"SetRate"		
"GetPreferredRate"	Rate	<p>These commands allows you to get and set the default movie playback speed. Normal speed is 65536, 1/2 speed (slow motion) is 32768, double speed is 131072. The example below restores the default playback speed (for example, after you have set the speed to slow motion. (Note: Pressing the Play button on the movie controller automatically sets the playback speed to the preferred speed).</p> <pre>local mSpeed superobject "MyMovie", "GetPreferredRate", mSpeed superobject "MyMovie", "SetRate", mSpeed</pre>
"SetPreferredRate"		
"GetTime"	Position	<p>These commands allows you to get and set the current movie playback location. You can use these commands to implement "bookmarks" within a movie. The value returned by the GetTime command is a binary value that represents the location within the movie. It is not a number (# of seconds, etc.) and cannot be used in calculations within Panorama. However, you can pass this value to the SetTime command to move the movie back to this location later. The example below shows two procedures for creating and using bookmarks within a movie. The first procedure adds a "bookmark" recording the current spot within the movie.</p> <pre>/* Procedure 1: Add a bookmark */ global movieMarks define movieMarks, "" local markName, markSpot markName="" gettext "Bookmark Name", markName superobject "MyMovie", "GetTime", markSpot movieMarks=sandwich("", movieMarks, ¶)+ markName+chr(9)+radixstr("hex", markSpot)</pre> <p>The second procedure is designed to be used with a pop-up menu or list superobject. When the user selects a bookmark in the menu or list the movie jumps to that spot.</p> <pre>/* Procedure 2: Goto a bookmark */ global movieMarks, movieSpot /* assume movieSpot contains name of bookmark, perhaps from pop-up menu */ local movieMark, markSpot, mNum mNum=arraysearch(movieMarks, movieSpot+chr(9)+"*", 1, ¶) movieMark=array(movieMarks, mNum, ¶) movieMark=array(movieMark, 2, chr(9)) markSpot=radix("hex", movieMark) superobject "MyMovie", "SetTime", markSpot</pre>
"SetTime"		

Command	Parameters	Description
"GetVolume"	Level	<p>These commands allows you to get and set the current movie playback volume. Full volume is 255, zero volume is 0. The example below cuts the current playback volume in half.</p> <pre>local mVol superobject "MyMovie", "GetVolume", mVol superobject "MyMovie", "SetVolume", mVol/2</pre>
"SetVolume"		
"GetPreferred Volume"	Level	<p>This command allows you to get the default movie playback volume. Full volume is 255, zero volume is 0. The example below sets the current playback volume to 1/3 of the default.</p> <pre>local mVol superobject "MyMovie", "GetPreferredVolume", mVol superobject "MyMovie", "SetVolume", mVol/3</pre>
"Play"		<p>This command starts the movie playing from the current location.</p> <pre>superobject "MyMovie", "Play"</pre>
"Stop"		<p>This command stops the movie playing.</p> <pre>superobject "MyMovie", "Stop"</pre>
"GoToBeginning"		<p>This command resets the current location to the beginning of the movie.</p> <pre>superobject "MyMovie", "GoToBeginning"</pre>
"GoToEnd"		<p>This command resets the current location to the end of the movie.</p> <pre>superobject "MyMovie", "GoToEnd"</pre>
"PlayFinished"		<p>This command checks to see if the movie has reached the end. The example starts the movie playback. If the movie is already at the end, the procedure resets the movie to the beginning before beginning playback.</p> <pre>local mStatus superobject "MyMovie", "PlayFinished", mStatus if mStatus=1 superobject "MyMovie", "GoToBeginning" endif superobject "MyMovie", "Play"</pre>

Super Flash Art Internal Data

This table describes the internal data in a Super Flash Art SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Super Flash Art™ Options](#)” on page 786.

Identifier	Data Type	Changeable?	Description
"#SUPER FLASH ART FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — scroll bars, borders, grow box, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#VERTICAL SCROLL BAR"	Bit	Yes	-1 if vertical scroll bar is enabled, 0 if disabled.
"#HORIZONTAL SCROLL BAR"	Bit	Yes	-1 if horizontal scroll bar is enabled, 0 if disabled.
"#INCLUDE PICTURES ON DISK"	Bit	Yes	-1 if include pictures on disk is enabled, 0 if disabled.
"#DISPLAY GROUP OF PICTURES"	Bit	Yes	-1 if display group of pictures is enabled, 0 if disabled.
"#TOP BORDER"	Bit	Yes	-1 if top border is enabled, 0 if disabled.
"#LEFT BORDER"	Bit	Yes	-1 if left border is enabled, 0 if disabled.
"#BOTTOM BORDER"	Bit	Yes	-1 if bottom border is enabled, 0 if disabled.
"#RIGHT BORDER"	Bit	Yes	-1 if right border is enabled, 0 if disabled.
"#GROW BOX"	Bit	Yes	-1 if grow box is enabled, 0 if disabled.
"#OVERFLOW PRINTING"	Bit	Yes	-1 if overflow is enabled, 0 if disabled (see “ Printing Multiple Page Records ” on page 1114 of the <i>Panorama Handbook</i>).
"#DROP SHADOW DEPTH"	Byte	Yes	0 if drop shadow is disabled. Non zero values specify the drop shadow offset (standard depth is 2 pixels).
"#SUPER FLASH ART ALIGNMENT"	Byte	Yes	Align option (see “ Align ” on page 797 of the <i>Panorama Handbook</i>). 0 = upper left 1 = upper center 2 = upper right 3 = middle left 4 = middle center 5 = middle right 6 = bottom left 7 = bottom center 8 = bottom right 9 = scale to fit 10 = scale to fit (proportional) 11 = tile
"#FLASH ART FILE"	Text	Yes	Alt File (tells Panorama to look in another database for Flash Art scrapbook, see “ Alt File ” on page 788 of the <i>Panorama Handbook</i>).
"#FLASH ART DEFAULT CAPTION"	Text	Yes	Default image name (see “ Default ” on page 787 of the <i>Panorama Handbook</i>).
"#FORMULA"	Text	No	Field name, variable name, or formula. If you want to be able to change the formula on the fly see “ Formula ” on page 786 of the <i>Panorama Handbook</i> .

Converting Between Image Formats

If the optional **Enhanced Image Pack** is installed a procedure can convert an image file from one format into another (for example from PICT into JPEG or JPEG into TIFF). (The **Enhanced Image Pack** requires that Apple Quicktime 4.0 or later be installed on your computer. If Quicktime is not already installed on your system you can download it from www.apple.com. It is also included on the Panorama CD.) Image conversions are performed with the `convertimage` statement.

```
convertimage input,output,type,height,width
```

The `input` parameter specifies the original image file. If the image file is in the same folder as the currently active database then only the file name is required (for example "`Cool Sunset.jpg`"). If the image file is in a different folder then both the folder and file name must be included (for example "`D:\Photography\Cool Sunset.jpg`"). (Note: The original image file may be a GIF file, but `convertimage` cannot produce a GIF output file.)

The `output` parameter specifies the new, converted image file. If you want to put this new image file in the same folder as the current database then only the file name is required, if you want to put it in a different folder then both the folder and file name must be included. If a file with this name already exists in this location it will be erased.

The `type` parameter specifies the type of image that will be created. If the output file has an extension (for example `.jpg`, `.pct`, `.tif`) you should leave this parameter blank ("") and let Panorama automatically figure out the type. If the output file does not have an extension you must specify the type from the list below.

Image Type	PC Extensions	Notes
PICT	.pct	Apple PICT bitmap
BMP	.bmp	Windows and OS/2 bitmap
JPEG	.jpg .jpeg	JPEG compressed image
PNG	.png	Portable Network Graphics bitmap
TIFF	.tif .tiff	Tagged Image Format
PHOTOSHOP	.psb	Adobe Photoshop
FLASHPIX	.fpx	FlashPix bitmap
TARGA	.targa	

The `height` and `width` parameters are the height and width of the new image (in pixels). If either (or both) of these parameters is zero then the height and/or width of the original image will be used.

Here is an example that converts a BMP image into a TIFF image. Since the output file has an extension (`.tif`) the output image type does not need to be specified. The TIFF image will have the same dimensions as the original PICT image.

```
convertimage "my picture.bmp","my picture.tif","",0,0
```

This example converts an image into a 32 by 32 pixel icon. Since the files do not have any extensions this example can only work on a Macintosh, not on a Windows PC.

```
convertimage "my picture","my icon","PICT",32,32
```

Here is a similar example that can work on a Windows PC (it can work on a Macintosh also, if the file names have extensions).

```
convertimage "my picture.jpg","my icon.jpg","",32,32
```

When the output file is in JPEG format you can use the `imagequality` statement to control the compression level of the JPEG conversion. This statement has one parameter, a number from 0 (very low quality, high compression) to 100 (high quality, least compression).

```
imagequality level
```

The `imagequality` statement must be used just before the `convertimage` statement. Here is an example that creates two JPEG images from a TIFF original, one low quality and one high quality. Each is placed in a different subfolder of the current database folder.

```
imagequality 80
convertimage "Sunset.tif",":Hi Quality:Sunset.jpg","",0,0
imagequality 20
convertimage "Sunset.tif",":Lo Quality:Sunset.jpg","",0,0
```

For more information on ordering the **Enhanced Image Pack** visit our website at <http://www.provue.com>.

Working with JPEG Images

Panorama has some statements specifically for working with JPEG images. The `jpegdimensions` statement will calculate the dimensions (height and width) of a JPEG image on the disk.

```
jpegdimensions path,height,width
```

The `path` parameter is the path and filename of the jpeg image to be measured, for example `MyDisk:MyFolder:Star.jpg`.

The `height` and `width` parameters specify variables to receive the dimensions of the image (in pixels).

The `bestfitjpeg` procedure copies a JPEG image. In the process of copying the file the procedure also resizes it to fit within a rectangle.

```
bestfitjpeg input,output,boundary
```

The `input` parameter specifies the original image file. If the image file is in the same folder as the currently active database then only the file name is required (for example `"Cool Sunset.jpg"`). If the image file is in a different folder then both the folder and file name must be included (for example `"D:\Photography\Cool Sunset.jpg"`).

The `output` parameter specifies the new, converted image file. If you want to put this new image file in the same folder as the current database then only the file name is required, if you want to put it in a different folder then both the folder and file name must be included. If a file with this name already exists in this location it will be erased.

The `boundary` parameter specifies the maximum dimensions of the copied image. The `bestfitjpeg` statement will make the image as large as possible without distorting the image within the rectangle.

This example will make a 48 by 48 pixel thumbnail of an image of a flower.

```
bestfitjpeg "Flower.jpg","Flower.Tiny.jpg",rectanglesize(0,0,48,48)
```

Taking an iSight Snapshot

If your MacOS X computer has an **iSight** camera a Panorama procedure can take a picture and save it to disk.

```
isightsnapshot folder,file,options
```

The **folder** parameter is the folder to save the snapshot in, or "" to save the snapshot in the same folder as the current database.

The **file** parameter specifies the name of the new snapshot (the file name). If this parameter is a variable that contains "" or does not contain any value Panorama will automatically assign a unique name and return it into this variable.

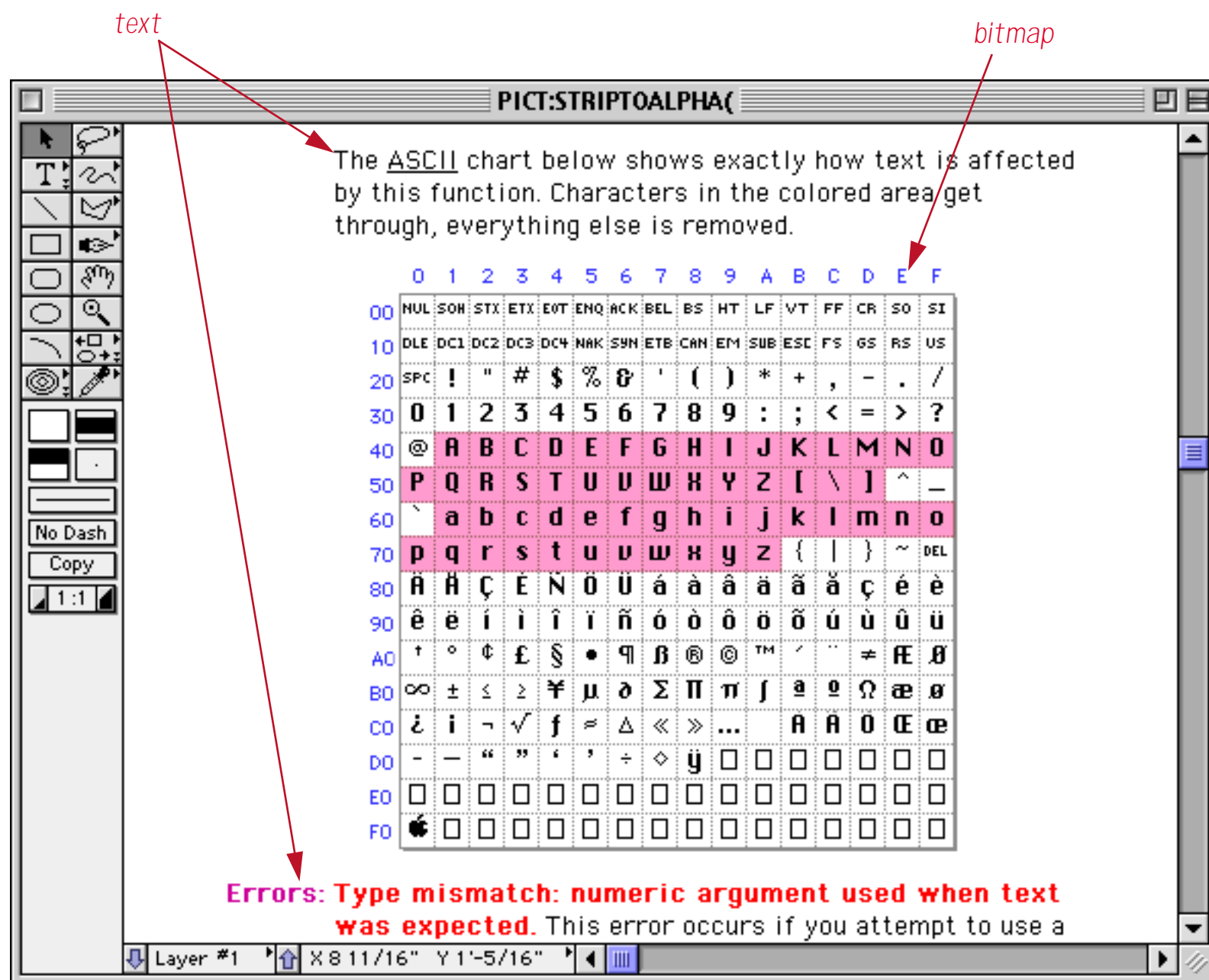
The **options** parameter is optional, and can be used to specify the type and size of the snapped image. The options parameter may contain a format specification (**format=jpg**, **format=png**, **format=tiff**, or **format=bmp**) and/or a scale (**scale=100%**, **scale=25%**, etc.). If no format is specified the statement will look at the filename to see if it can figure out the image type (ending with .jpg, .tiff, .bmp, etc.). If all else fails it will default to jpeg. If no scale is specified the default is 100% (640 by 480). This example takes a 1/2 size snapshot and saves it as a JPEG file.

```
isightsnapshot "Say Cheese.jpg","scale=50%"
```

This image can now be displayed using Panorama's Super Flash Art feature (assuming you have the Enhanced Image Pack installed).

Building Web Like HyperText Systems with Super Flash Art

Apple's PICT image format allows an image to contain text as well as bitmap information. For example the image below (shown in Deneba's Canvas 3.5) contains both text and bitmap graphics.



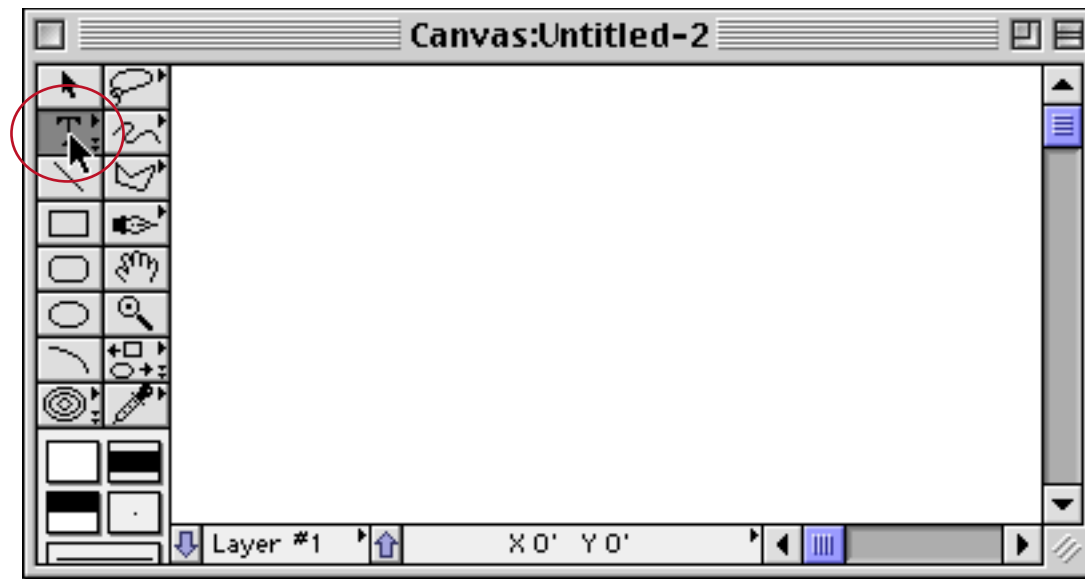
Depending on how the picture was created, it can be possible to extract the text in a picture based on certain specifications: location, style, color, font, etc. This feature gives Panorama the power to turn a collection of pictures into a linked hypertext system. The **Panorama On-Line Reference** is an example of such a system. This system is basically just a collection of PICT images. When the user clicks on a word or phrase within an image (for example the word **ASCII** in the image above) a simple procedure decodes what word or phrase they clicked on and switches to the new page (a new picture) based on that information.

Preparing Pictures with Extractable Text

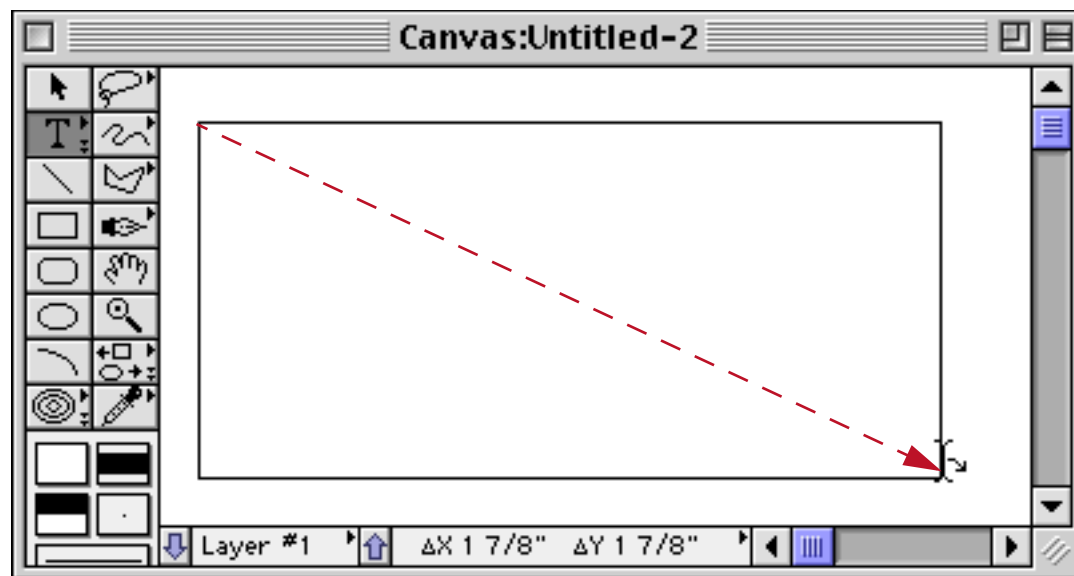
Not all text in every picture can be extracted. Text that has been converted to a pixel (bitmap) image cannot be extracted. As a general rule, if the text can be edited as text in your drawing program, the text can be extracted. For example, Photoshop does not allow text to be edited after it has been created, so text in an image created by Photoshop cannot be extracted. (Of course, Photoshop does allow the text to be manipulated with graphic tools, but that doesn't count. You must be able to insert and delete text, type in new text, etc.)

Some programs that work well for creating extractable text include Canvas and Freehand. (In Canvas, you must make sure that the text is in a text object, not a paint object.) For other programs we recommend that you try a small picture before you do a lot of work. Our favorite program for creating images with extractable text is **Deneba Canvas** (see <http://www.deneba.com>).

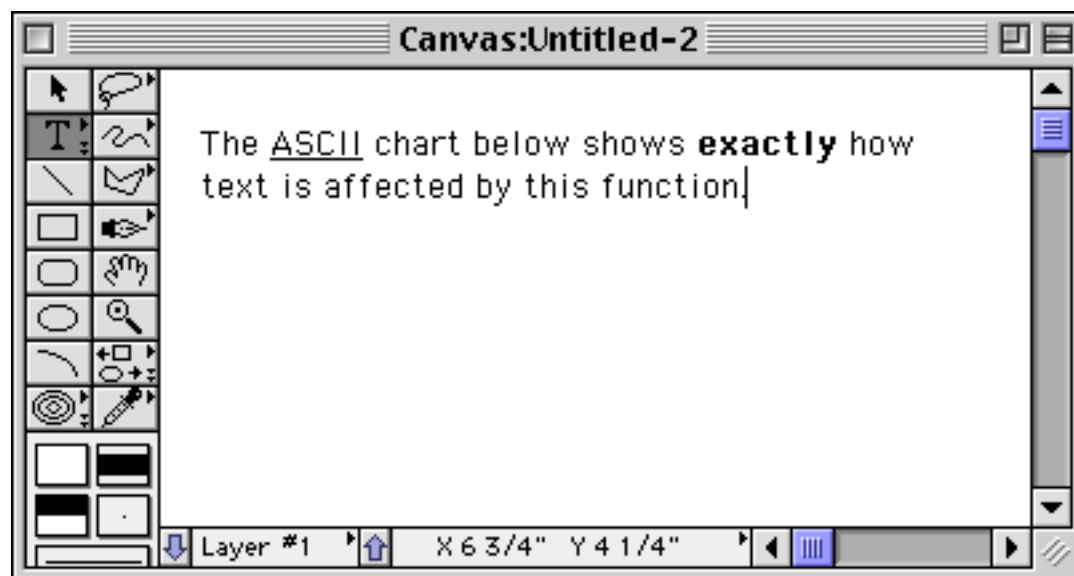
To create extractable text with Canvas you start by selecting the Text tool.



Next, you drag the mouse over the location where you want to create the text, just like creating auto-wrap text in Panorama.



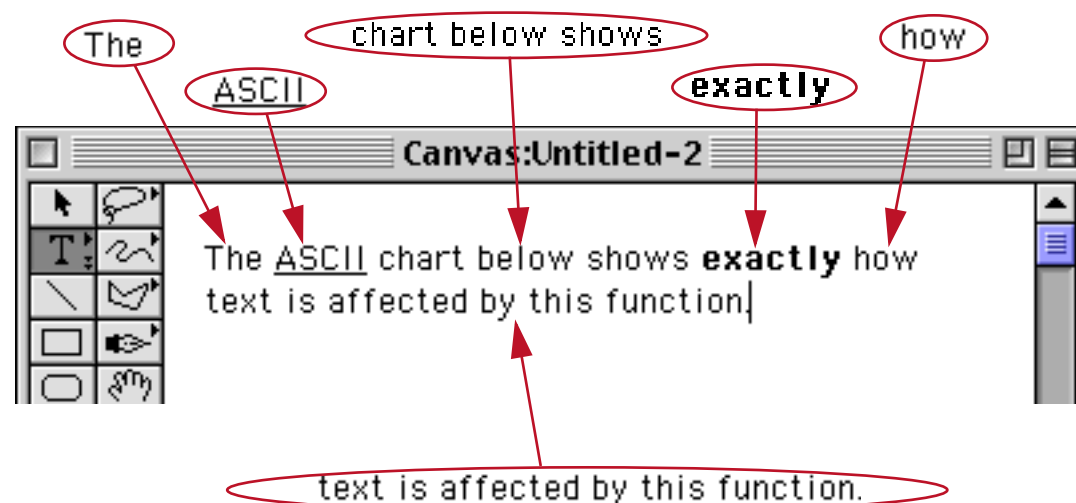
Now type in the text.



When the image is complete, be sure to save it using PICT format.

Programming a HyperText Engine

When text is saved as part of an image in PICT format the text is split up internally into chunks. A new chunk starts: 1) whenever there is any change in the text font, size, style or color, or 2) whenever a new line begins. The image created in the previous section actually consists of six separate text chunks.



The Super Flash Art **"FindText"** command takes a point within the image (x,y co-ordinates) and checks to see if a chunk of text is at that location. If there is, it extracts the text chunk and returns it to the procedure for further processing. Using this command a procedure can find out what chunk of text has been clicked on (if any) and take appropriate action.

Finding out what chunk of text has been clicked on takes three components: 1) a Super Flash Art object (see ["Creating Super Flash Art Objects"](#) on page 751 of the *Panorama Handbook*), 2) a transparent "Classic" Pushbutton with the click/release option turned off (see ["Transparent Push Buttons"](#) on page 832 of the *Panorama Handbook*), and 3) a procedure triggered by the "Classic" pushbutton. The transparent button should be overlaid exactly on top of the Super Flash Art object...use the **Align** command to get exact alignment (see ["Aligning Objects"](#) on page 553 of the *Panorama Handbook*). If the Super Flash Art object has scroll bars, however, they should not be covered by the button. Only cover the area where the actual image is displayed.

The example below shows a procedure that will figure out what text was clicked on, and what font, size, and style the text is. The example assumes that the Super Flash Art object is named **HyperFlash**. (To give an object a name, first select the object, then use the **Object Name** command in the Edit menu or click on the object name in the Graphic Control Strip, see ["Object Type/Object Name"](#) on page 533 of the *Panorama Handbook*.)

```
local v,h,clickPoint
local clickText, clickFont, clickSize, clickStyle
v=v(info("mouse"))-rtop(info("buttonrectangle"))
h=h(info("mouse"))-rleft(info("buttonrectangle"))
clickPoint=point(v,h)
superobject "HyperFlash","FindText",clickPoint,clickText
clickFont=objectinfo("font")
clickSize=objectinfo("textsize")
clickStyle=objectinfo("textstyle")
```

This example uses the special **superobject "FindText"** command. This command only works with Super Flash Art objects. The command has two additional parameters: 1) the location within the picture object (in this case **clickPoint**) and the field or variable the extracted text should be placed into (in this case **clickText**).

If the click was over a chunk of text, the "FindText" command will extract that chunk. After the chunk has been extracted the procedure can use the `objectinfo()` function to find out the font, text size, and style of the chunk (as shown in the program above). The style is a number that is calculated by adding up the following numbers for each possible style:

0	Plain
1	Bold
2	Italic
4	Underline
8	Outline
16	Shadow

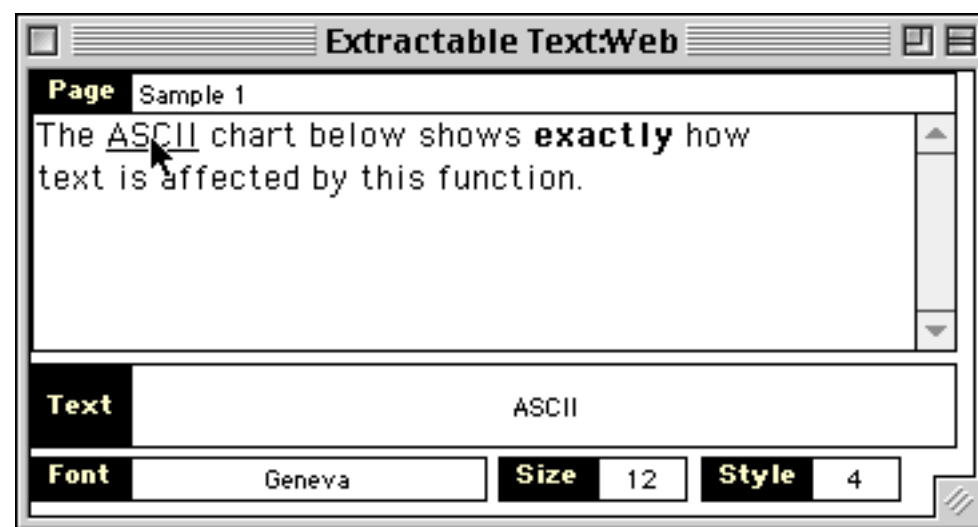
For example, if the chunk is bold-italic, the `objectinfo("textstyle")` function will return the value 3.

The statements shown below could be added to the end of the previous program to ignore all text that is not underlined.

```
if (clickStyle and 4) <> 4
  stop
endif
```

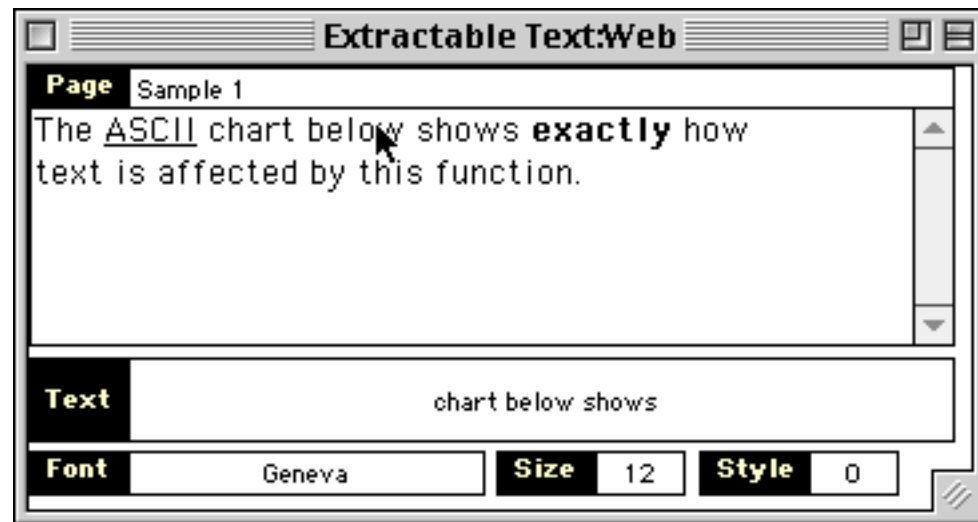
The `and` operator isolates only the underline attribute. If the statement was simply `if clickStyle<>4` the procedure would stop if the chunk was a combination style like bold-underline or italic-underline.

The sample database **Extractable Text** demonstrates Panorama's ability to extract chunks of text. When you click on the image it uses the procedure listed above to extract the text, along with its font, size, and style. For example, if you click on the word ASCII as shown here it extracts and displays the chunk of text (ASCII) along with all of the attributes of the chunk.



As you can see the chunk of text is **ASCII**, the font is **Geneva 12**, and the style is **bold (4)**.

You can click on any location to see what chunk of text is there.



If there is no chunk of text at the spot that was clicked the `clickText` variable will contain empty text ("").

Extracting All Text of a Specific Style

The `superobject "ExtractText"` command allows all the text that matches specified criteria to be extracted from the picture currently displayed in a Super Flash Art object. This command has several parameters as shown here:

```
SuperObject "Object Name", "ExtractText", Font, Size, Style, Sep, Result
```

The first parameter, **Font**, specifies the font of the text you want to extract. If you don't care what the font is, leave this parameter empty ("").

The second parameter, **Size**, specifies the size of the text you want to extract. If you don't care, this parameter should be zero.

The third parameter, **Style**, specifies the style of the text you want to extract. This parameter allows you extreme flexibility in selecting what styles you want to extract.

If the **Style** parameter is zero, any style is ok. For example, if you want to extract all **Monaco 9** point text of any style into a variable named **Samples**, here's how you would do it:

```
local Samples
superobject "HyperFlash", "ExtractText", "Monaco", 9, 0, ";", Samples
```

If the **Style** parameter is 1-255, it specifies the exact style you want. Add up the numbers for each individual style. For example, for underlined text you would specify **4**, for bold text, **1**. The example below will extract all bold text, but not bolditalic or bold underlined.

```
local Samples
superobject "HyperFlash", "ExtractText", "", 0, 1, ¶, Samples
```

If the **Style** parameter is 256 or greater, it specifies both the style and a mask for the style. The mask allows you to isolate individual styles. The mask uses the same style numbers as the individual styles, but multiplied by 256. (Why 256? 256 is 2 the 8th power (2^8), an even number in the computer's binary numbering system.) For example, suppose you wanted to extract all bold text, even bold text that is combined with other styles. By using a mask of $4*256$ you tell the `"ExtractText"` command that you only care about the underlined style. The example below will extract all underlined, underlined-italic, underlined-bold; any text that is underlined no matter what other attributes it may have.

```
local Samples
superobject "HyperFlash", "ExtractText", "", 0, (4*256)+4, ¶, Samples
```

The fourth parameter, **Sep**, specifies what separator character(s) should be used between text chunks as they are extracted. Usually this is a carriage return (¶), comma, space, slash, etc. (see “[Picking a Separator Character](#)” on page 93).

The character "t" is a special separator. When this separator is used, Panorama checks each piece of extracted text to see if it is on the same line as the previously extracted piece of text. If it is on the same line, Panorama will connect the pieces with a space. If the two pieces are on different lines, they will be connected with a carriage return. This allows the extracted text to be more or less reconstructed in its original form. (Note: Either "t" or "T" will trigger this special operation.)

The final parameter, **Result**, is the field or variable that you want the extracted text placed into.

The procedure below displays the number of underlined segments in the current picture.

```
local KeywordList
superobject "HyperFlash", "ExtractText", "", 0, 4, ¶, KeywordList
message arraysize(KeywordList, ¶)
```

The next example takes all the Monaco 9 pt text in the current picture and combines it. It then copies the text onto the clipboard.

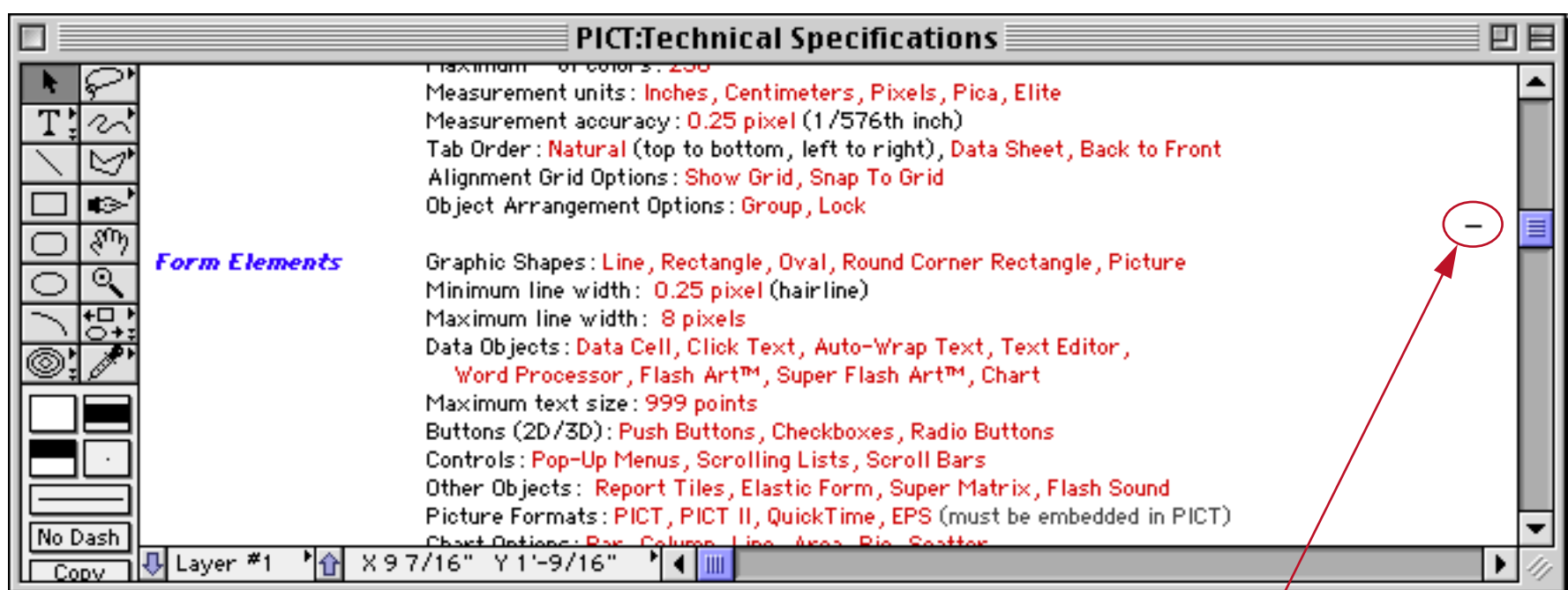
```
local SampleText
superobject "HyperFlash", "ExtractText", "Monaco", 9, 0, "t", SampleText
clipboard=SampleText
```

Creating Multi-Page Pictures

Most pictures fit on a single page. However, if you are creating a HyperText system, you may want to build pictures that are several pages long. On the screen, the user can use scroll bars to see the entire picture. But what about printing?

To allow printing of multi-page text or pictures, Panorama includes a feature called an “overflow” tile. The “overflow” tile works in conjunction with a regular data tile to print any leftover data that would not fit on the data tile. See “[Printing Data that Overflows a Page](#)” on page 1116 of the *Panorama Handbook* to learn how to set up multi-page printing.

If you use this picture overflow system, you can build special objects into your pictures that will tell Panorama where to split the picture into separate pages when printing. This will prevent the printout from splitting the page in the middle of a graphic or in the middle of a line of text. To specify a page break, you must draw a short horizontal line (a few pixels wide) near the right edge of the picture (within 16 pixels of the right edge) as shown in this example.



short horizontal line forces page break

Push Button Internal Data

This table describes the internal data in a Push Button SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Super Object Push Button](#)” on page 823.

Identifier	Data Type	Changeable?	Description
"#PUSH BUTTON FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — 3D text, click/release, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#CLICK/RELEASE"	Bit	Yes	-1 if click/release is enabled, 0 if disabled.
"#3D TEXT"	Bit	Yes	-1 if 3D text is enabled, 0 if disabled.
"#HIDE BUTTON TITLE"	Bit	Yes	-1 if hide title is enabled, 0 if disabled.
"#COLOR TITLE"	Bit	Yes	-1 if color:title is enabled, 0 if disabled.
"#COLOR BORDER"	Bit	Yes	-1 if color:border is enabled, 0 if disabled.
"#COLOR FILL"	Bit	Yes	-1 if color:fill is enabled, 0 if disabled.
"#COLOR HIGHLIGHT"	Bit	Yes	-1 if color:highlight is enabled, 0 if disabled.
"#PUSH BUTTON STYLE"	Byte	Yes	Style option (see “ Push Button Styles ” on page 826 of the <i>Panorama Handbook</i>). -1 = transparent 0 = plain rectangle 1 = plain rounded rectangle 2 = plain circle 3 = Standard Push Button (default) 4 = 3D rectangle 5 = 3D rounded rectangle 6 = 3D circle 7 = Beveled Rectangle
"#BUTTON TITLE OFFSET"	Byte	Yes	+/- vertical title offset (see “ Title Positioning ” on page 828 of the <i>Panorama Handbook</i>).
"#BUTTON TITLE"	Text	Yes	Title of button (maximum 31 characters). The example procedure below changes a button’s title from Go to Stop to Go to Stop each time the procedure runs. <pre> local bTitle selectobjects objectinfo("name")="Signal" bTitle=objectinfo("#BUTTON TITLE") if bTitle="Go" bTitle="Stop" else bTitle="Go" endif changeobjects "#BUTTON TITLE",bTitle selectnoobjects </pre>
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this button is pressed.

Flash Art Push Button Internal Data

This table describes the internal data in a Flash Art Push Button SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Flash Art™ Push Button SuperObjects™](#)” on page 833.

Identifier	Data Type	Changeable?	Description
"#PUSH BUTTON FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — click/release, include pictures on disk, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#CLICK/RELEASE"	Bit	Yes	-1 if click/release is enabled, 0 if disabled.
"#INCLUDE PICTURES ON DISK"	Bit	Yes	-1 if include pictures on disk is enabled, 0 if disabled.
"#BUTTON TITLE"	Text	Yes	Title of button (maximum 31 characters).
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this button is pressed.
"#FLASH ART FILE"	Text	Yes	Alt File (tells Panorama to look in another database for Flash Art scrapbook, see “ Alt File ” on page 788 of the <i>Panorama Handbook</i>).
"#FORMULA"	Text	No	Formula used to select which flash art image to display.

Data Button SuperObject Internal Data

This table describes the internal data in a Data Button SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Super Data Button Options](#)” on page 849.

Identifier	Data Type	Changeable?	Description
"#DATA BUTTON FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — allow multiple values, “radio” button, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#MULTIPLE VALUES"	Bit	Yes	-1 if allow multiple values is enabled, 0 if disabled.
"#RADIO BUTTON"	Bit	Yes	-1 if “ radio ” button is enabled, 0 if disabled.
"#BUTTON TITLE"	Text	Yes	Title of button (maximum 49 characters).
"#SEPARATOR"	Text	Yes	Value separator for multiple values (maximum 5 characters).
"#BUTTON ON VALUE"	Text	Yes	Value of button (maximum 49 characters).
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this button is pressed.
"#FORMULA"	Text	No	Name of field or variable that will contain data value.

Flash Art Data Button SuperObject Internal Data

This table describes the internal data in a Flash Art Data Button SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Flash Art Data Button SuperObjects™](#)” on page 852.

Identifier	Data Type	Changeable?	Description
"#FLASH STICKY BUTTON FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — allow multiple values, “radio” button, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#CLICK/RELEASE"	Bit	Yes	-1 if click/release is enabled, 0 if disabled.
"#BUTTON ANIMATION"	Bit	Yes	-1 if f/x is enabled, 0 if disabled.
"#HIDE BUTTON TITLE"	Bit	Yes	-1 if hide title is enabled, 0 if disabled.
"#INCLUDE PICTURES ON DISK"	Bit	Yes	-1 if include pics on disk is enabled, 0 if disabled.
"#MULTIPLE VALUES"	Bit	Yes	-1 if allow multiple values is enabled, 0 if disabled.
"#RADIO BUTTON"	Bit	Yes	-1 if “ radio ” button is enabled, 0 if disabled.
"#BUTTON TITLE OFFSET"	Byte	Yes	+/- vertical title offset (see “ Title Positioning ” on page 828 of the <i>Panorama Handbook</i>).
"#BUTTON TITLE"	Text	Yes	Title of button (maximum 49 characters).
"#SEPARATOR"	Text	Yes	Value separator for multiple values (maximum 5 characters).
"#BUTTON ON VALUE"	Text	Yes	Value of button (maximum 49 characters).
"#FLASH ART FILE"	Text	Yes	Alt File (tells Panorama to look in another database for Flash Art scrapbook, see “ Alt File ” on page 788 of the <i>Panorama Handbook</i>).
"#FLASH ART FORMULA"	Text	No	Formula used to select which flash art image to display.
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this button is pressed.
"#FORMULA"	Text	No	Name of field or variable that will contain data value.

Sticky Push Button SuperObject Internal Data

This table describes the internal data in a Sticky Push Button SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Sticky Push Button SuperObjects™](#)” on page 855.

Identifier	Data Type	Changeable?	Description
"#STICKY PUSH BUTTON FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — allow multiple values, “radio” button, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#CLICK/RELEASE"	Bit	Yes	-1 if click/release is enabled, 0 if disabled.
"#3D TEXT"	Bit	Yes	-1 if 3D text is enabled, 0 if disabled.
"#HIDE BUTTON TITLE"	Bit	Yes	-1 if hide title is enabled, 0 if disabled.
"#MULTIPLE VALUES"	Bit	Yes	-1 if allow multiple values is enabled, 0 if disabled.
"#RADIO BUTTON"	Bit	Yes	-1 if “ radio ” button is enabled, 0 if disabled.
"#COLOR TITLE"	Bit	Yes	-1 if color:title is enabled, 0 if disabled.
"#COLOR BORDER"	Bit	Yes	-1 if color:border is enabled, 0 if disabled.
"#COLOR FILL"	Bit	Yes	-1 if color:fill is enabled, 0 if disabled.
"#COLOR HIGHLIGHT"	Bit	Yes	-1 if color:highlight is enabled, 0 if disabled.
"#PUSH BUTTON STYLE"	Byte	Yes	Style option (see “ Push Button Styles ” on page 826 of the <i>Panorama Handbook</i>). 0 = rectangle 1 = rounded rectangle 2 = circle 3 = 3D rectangle 4 = 3D rounded rectangle 5 = 3D circle 6 = Beveled Rectangle
"#BUTTON TITLE OFFSET"	Byte	Yes	+/- vertical title offset (see “ Title Positioning ” on page 828 of the <i>Panorama Handbook</i>).
"#BUTTON TITLE"	Text	Yes	Title of button (maximum 49 characters).
"#SEPARATOR"	Text	Yes	Value separator for multiple values (maximum 5 characters).
"#BUTTON ON VALUE"	Text	Yes	Value of button (maximum 49 characters).
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this button is pressed.
"#FORMULA"	Text	No	Name of field or variable that will contain data value.

Pop-Up Menu SuperObject Internal Data

This table describes the internal data in a Pop-Up Menu SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Pop-Up Menu Options](#)” on page 866.

Identifier	Data Type	Changeable?	Description
"#POP-UP MENU FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — multi-column menus, combo box, show value, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#POP-UP 2 PIXEL DROP SHADOW"	Bit	Yes	-1 if drop shadow:2 pixels is enabled, 0 if disabled.
"#POP-UP 1 PIXEL DROP SHADOW"	Bit	Yes	-1 if drop shadow:1 pixel is enabled, 0 if disabled.
"#POP-UP SHOW VALUE"	Bit	Yes	-1 if show value is enabled, 0 if disabled.
"#POP-UP CHICAGO"	Bit	Yes	-1 if Chicago 12 is enabled, 0 if disabled.
"#POP-UP MENU WRAPPING"	Bit	Yes	-1 if multi-column is enabled, 0 if disabled.
"#POP-UP TRIANGLE"	Bit	Yes	-1 if show triangle is enabled, 0 if disabled.
"#COMBO BOX"	Bit	Yes	-1 if combo box is enabled, 0 if disabled.
"#SMART POP UP COMBO BOX"	Bit	Yes	-1 if mac popup/windows combo box is enabled, 0 if disabled.
"#POP-UP FILL COLOR"	Text	Yes	Fill color for pop-up menu (see “ Colors ” on page 154).
"#POP-UP LAST ITEM"	Word	No	Last menu item selected (number from 1 to maximum number of items in menu). For example if the menu contains four items (Red , Green , Blue , Orange) and the user picks Blue this value will be 3 .
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this button is pressed.
"#FIELD"	Text	Yes	Name of field or variable that contains value of pop-up menu (maximum 31 characters).
"#FORMULA"	Text	No	Formula for calculating contents of menu.

List SuperObject™ Commands

The List SuperObject understands about a dozen commands that can be sent to it with the `superobject` statement in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). This table describes each of these commands in detail.

Command	Parameters	Description
"FillList"	Formula,Database	<p>This command re-fills the specified list. You can use this command to update the list, or to fill it with completely new information.</p> <p>To update the list using the settings in the List Configuration Dialog (see “List Options” on page 883 of the <i>Panorama Handbook</i>) leave off the Formula and Database. For example, if the pizza toppings list was derived from a pizza toppings database, you would want to use this procedure when the pizza toppings database had changed:</p> <pre>superobject "Toppings","FillList"</pre> <p>You can also use the "FillList" command to fill the list with entirely new information, completely ignoring the formula and database originally specified in the List dialog. The same list can be filled and refilled again and again with different items as conditions change. Below are three samples that could be used to fill a list from the Pizza Toppings database. The first sample lists all toppings, the next veggie only, the final meat only.</p> <pre>superobject "Toppings","FillList", Topping,"Pizza Toppings" superobject "Toppings","FillList", ?(Category="Veggie",Topping,""),"Pizza Toppings" superobject "Toppings","FillList", ?(Category="Meat",Topping,""),"Pizza Toppings"</pre> <p>Of course you can also use the "FillList" command to directly specify the contents of the list. In this case the database should be set to "".</p> <pre>superobject "Toppings","FillList", "Pepperoni"+¶+"Sausage"+¶+"Meatballs"+¶+ "Mushrooms"+¶+"Olives"+¶+"Onions" , ""</pre> <p>Of course the topping list could also be created with variables. Here’s an example:</p> <pre>local MeatToppings,VeggieToppings,SpecialtyToppings MeatToppings="Pepperoni"+¶+"Sausage"+¶+"Meatballs" VeggieToppings="Mushrooms"+¶+"Olives"+¶+"Onions" SpecialtyToppings="Anchovies"+¶+"Garlic" superobject "Toppings","FillList",VeggieToppings,""</pre>
"AutoScroll"		<p>This command scrolls the list so that the first selected item is visible. For example, this procedure selects Pineapple and scrolls the list to make sure that the Pineapple item is visible. (The procedure assumes that the selected list value is stored in a field named ListCell — see “Data” on page 883 of the <i>Panorama Handbook</i>).</p> <pre>ListCell="Pineapple" SuperObject "Toppings","AutoScroll"</pre>

Command	Parameters	Description
"CellRectangle"	Item,Rectangle	<p>This command allows a procedure to determine the physical location and size (i.e. rectangle) of any item in the list. This command has two parameters as shown below: the Item number (from 1 to the maximum number of items in the list) and the Rectangle. The Rectangle should be a variable that will contain the final result.</p> <p>Here is an example that fills in the variable <code>dragRectangle</code> with the dimensions of the third item in the list.</p> <pre>superobject "My List", "CellRectangle",3,dragRectangle</pre> <p>Note: The rectangle that is returned by this command is in window relative co-ordinates (see "Rectangles" on page 149). You can change this to screen or form relative co-ordinates using the <code>xytoxy()</code> function (see "XYTOXY()" on page 5910).</p>
"PointToCell"	Point,Cell	<p>This command allows a procedure to determine what list item (if any) corresponds to any point on the screen. For example, if someone drags something onto the list, this command allows the procedure to determine where in the list the item should be placed. This command has two parameters as shown below: the Point and the Cell. The Cell parameter should be a variable that will contain the final result.</p> <pre>superobject "object name","PointToCell",Point,Cell</pre>
"GetList"	List	This command produces a list of all the items in the list, with each item separated from the next by a carriage return. The list is placed into the field or variable specified by List .
"GetSelected"	List	This command produces a list of all the selected items in the list, with each item separated from the next by a carriage return. The list is placed into the field or variable specified by List . (Note: This command is redundant if the list is already associated with a field or variable. Instead of using the command the procedure can simply examine (or change!) the value of the field or variable.)
"GetCount"	Number	This command returns a count of the total number of items currently in the list into the field or variable specified by Number .
"GetCell"	Item,Value	<p>This command extracts the contents of a particular item in the list. The command treats the list as a series of numbered items, starting from 1 at the top of the list. This example will copy the first item in the list <code>PartsList</code> into the variable <code>NextPart</code>.</p> <pre>local NextPart superobject "PartsList","GetCell",1,NextPart</pre> <p>This example will copy the last item in the list <code>PartsList</code> into the variable <code>NextPart</code>.</p> <pre>local NextPart,ListCount superobject "PartsList","GetCount",ListCount superobject "PartsList", "GetCell",ListCount,NextPart</pre>

Command	Parameters	Description
"FindCell"	Cell,Text	<p>This command searches the list to find a specified value. The list item must match exactly, or the search will be unsuccessful. The search starts with the item specified by Cell. If successful, the number of the item containing the searched for value will be placed in Cell, otherwise Cell will be set to zero. The example below will locate Garlic in the list of pizza toppings and select it (tasty!).</p> <pre data-bbox="971 523 1954 749"> local ListCell ListCell=1 superobject "Toppings","FindCell",ListCell,"Garlic" if ListCell≠0 superobject "Toppings","SelectCell",ListCell endif </pre> <p>Keep in mind that the word or phrase must match exactly. In this case only Garlic will be located; garlic or Roasted Garlic will not.</p> <p>Note: This command is redundant if the list is already associated with a field or variable. Instead of using the "FindCell" command the procedure can simply set the value of the field or variable. Here is a much simpler procedure that performs the same function as the procedure above. (The procedure assumes that the selected list value is stored in a field named ListCell — see "Data" on page 883 of the <i>Panorama Handbook</i>).</p> <pre data-bbox="971 1131 1397 1201"> ListCell="Garlic" showvariables ListCell </pre>
"SelectCell"	Cell	<p>This command selects a specified item in the list. The item is specified by Cell, which should be a number from 1 to the maximum number of items in the list.</p>
"UnSelectCell"	Cell	<p>This command unselects a specified item in the list. The item is specified by Cell, which should be a number from 1 to the maximum number of items in the list. The example below makes sure that there are no anchovies on the pizza!</p> <pre data-bbox="971 1583 1911 1838"> local ListCell ListCell=1 SuperObject "Toppings","FindCell",ListCell,"Anchovies" if ListCell≠0 superobject "Toppings","UnSelectCell",ListCell endif </pre> <p>Note: This command is usually redundant if the list is already associated with a field or variable. Instead of using the "UnSelectCell" command the procedure can simply set the value of the field or variable. Here is a much simpler procedure that performs the same function as the procedure above. (The procedure assumes that the selected list value is stored in a field named ListCell that is a carriage return separated array — see "Data" on page 883 of the <i>Panorama Handbook</i>).</p> <pre data-bbox="971 2177 1703 2290"> ListCell=replace(ListCell,"Garlic","") ListCell=arraystrip(ListCell,¶) showvariables ListCell </pre>

Command	Parameters	Description
"SetCell"	Cell, Value	<p>This command changes the contents of a specified item in the list. The item is specified by Cell, and should be from 1 to the maximum number of items in the list. The example below changes the Cheese item to Extra Cheese.</p> <pre>local ListCell ListCell=1 superobject "Toppings", "FindCell", ListCell, "Cheese" if ListCell<>0 superobject "Toppings", "SetCell", ListCell, "Extra Cheese" endif</pre>
"AddCell"	Value	<p>This command adds a new item to the end of the list. This example adds the item Sun Dried Tomatoes to the end of the list of pizza toppings.</p> <pre>superobject "Toppings", "AddCell", "Sun Dried Tomatoes"</pre>
"InsertCell"	Cell, Value	<p>This command inserts a new item into the middle of the list. The Cell parameter specifies where the new item should be inserted. This parameter must be a number from 1 up to the number of items in the list. The new item will go above the item specified. For example, you could insert the item Extra Cheese at the very top of the pizza topping list:</p> <pre>superobject "Toppings", "InsertCell", 1, "Extra Cheese"</pre> <p>This more complex example inserts Grilled Onions after Onions.</p> <pre>local ListCell ListCell=1 superobject "Toppings", "FindCell", ListCell, "Onions" if ListCell≠0 ListCell=ListCell+1 superobject "Toppings", "InsertCell", ListCell, "Grilled Onions" endif</pre> <p>Notice that the example adds one to ListCell before inserting the new item. This is so the new item (Grilled Onions) will be inserted after the original item (Onions) instead of before it.</p>
"DeleteCell"	Start, End	<p>This command deletes one or more items from the list. If you just want to delete a single cell, then only one number is needed. This example deletes the first item in the list.</p> <pre>superobject "Toppings", "DeleteCell", 1</pre> <p>To delete a bunch of cells at once, specify two numbers — the first and last cell to delete. This example deletes the first 5 items in the list.</p> <pre>superobject "Toppings", "DeleteCell", 1, 5</pre> <p>This example will delete the entire list in a big hurry!</p> <pre>superobject "Toppings", "DeleteCell", 1, 10000</pre>

Command	Parameters	Description
"FindSelected"	Cell	<p>This command finds the next selected cell, starting with Cell. The result is placed in Cell, or zero if there are no selected cells below the starting spot. The example below deletes all the selected items from the list.</p> <pre> Local Spot Spot=1 loop SuperObject "Toppings", "FindSelected", Spot if Spot=0 stop endif SuperObject "Toppings", DeleteCell, Spot next </pre>

Using Drag and Drop to Change the Order of Items in a List

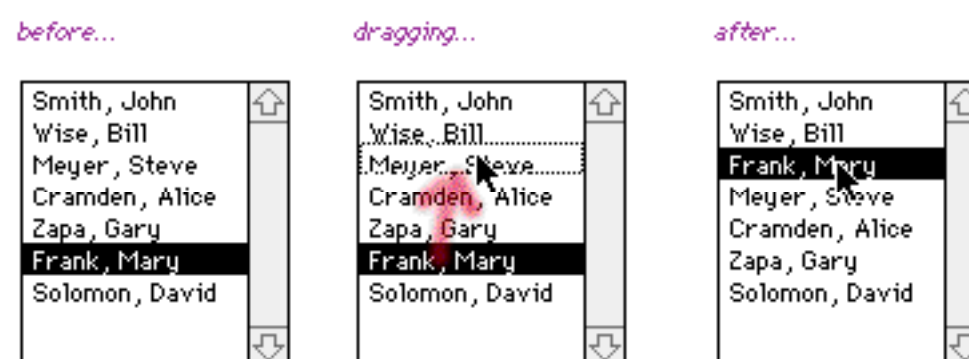
This example shows how to set up a procedure that allows the user to drag items up or down in a list to change the order of a list. This example assumes that there is a list of names in a field called **Names**, with each name separated from the next by a carriage return. The current form must contain a List SuperObject named **Names List** (see “[Object Type/Object Name](#)” on page 533 of the *Panorama Handbook*). The **Names List** object displays the **Names** field, it is also linked to a global variable named **theName** (see “[Data](#)” on page 883 of the *Panorama Handbook*).

```

global theName
local cell,cellbox,listbox,mouse,newcell,newNames
cell=1
superobject "Names List","findselected",cell
superobject "Names List","cellrectangle",cell,cellbox
cellbox=xytoxy(cellbox,"w","s")
object "Names List"
listbox=xytoxy( objectinfo("rectangle"),"f","s")
draggraybox cellbox,listbox,listbox,0
if cellbox="" stop endif /* user dragged out of the list */
mouse=xytoxy( info("mouse"),"s","w")
/* where is the new location for this item? */
superobject "Names List","pointtocell",mouse,newcell
if cell=newcell stop endif /* item did not move */
if newcell>cell
  newcell=newcell-1 /* adjust for deleting item in old spot */
endif
/* delete item from old spot */
newNames=arraydelete(Names,cell,1,1)
if newcell>0
  /* insert item in new spot */
  newNames=arrayinsert(newNames,newcell,1,1)
  newNames=arraychange(newNames,theName,newcell,1)
else
  /* add item to end of list */
  newNames=newNames+1+theName
endif
Names=newNames /* update original field */
superobject "Names List","filllist" /* re-display list */
showvariables theName /* and select correct item */

```


The illustration below shows this procedure in action. You can click on any item in the list and drag it into a new position on the list.



List SuperObject Internal Data

This table describes the internal data in a List SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[List Options](#)” on page 883.

Identifier	Data Type	Changeable?	Description
"#LIST FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — sort up, no duplicates, click/release, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#LIST SORT"	Bit	Yes	-1 if sort up is enabled, 0 if disabled.
"#LIST NO DUPLICATES"	Bit	Yes	-1 if no duplicates is enabled, 0 if disabled.
"#GROW BOX"	Bit	Yes	-1 if grow box is enabled, 0 if disabled.
"#CLICK/RELEASE"	Bit	Yes	-1 if click/release is enabled, 0 if disabled.
"#THIN SCROLL"	Bit	Yes	-1 if thin scroll bars are enabled, 0 if disabled.
"#LIST CLICK FLAGS"	Byte	Yes	This value controls the click action configuration (see “ Click Action ” on page 892 of the <i>Panorama Handbook</i>). 0 = Normal 128 = One Cell Only 32 = Contiguous Cells Only 118 = Extend w/o Shift
"#LIST DATABASE"	Text	Yes	Name of database to scan ("" if formula builds list directly).
"#SEPARATOR"	Text	Yes	Value separator for multiple values (maximum 5 characters).
"#PROCEDURE"	Text	Yes	Name of the procedure that is triggered when this list is pressed.
"#FIELD"	Text	Yes	Name of field or variable that contains value of pop-up menu (maximum 31 characters).
"#FORMULA"	Text	No	Formula for calculating contents of list.

Auto Grow SuperObject™ Commands (Elastic Forms)

The Auto Grow SuperObject (see “[Elastic Forms](#)” on page 922 of the *Panorama Handbook*) understands a small set of commands that can be sent to it with the `superobject` statement in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). This table describes each of these commands in detail.

Command	Parameters	Description
"GetMinSize"	Height,Width	This command gets the minimum window size (see “ Building an Elastic Form ” on page 925 of the <i>Panorama Handbook</i>).
"SetMinSize"	Height,Width	This command sets the minimum window size (see “ Building an Elastic Form ” on page 925 of the <i>Panorama Handbook</i>).
"GetMaxSize"	Height,Width	This command gets the maximum window size (see “ Maximum Window Size ” on page 929 of the <i>Panorama Handbook</i>).
"SetMaxSize"	Height,Width	This command sets the maximum window size (see “ Maximum Window Size ” on page 929 of the <i>Panorama Handbook</i>).

Auto Grow SuperObject Internal Data

This table describes the internal data in an Auto Grow SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Building an Elastic Form](#)” on page 925 of the *Panorama Handbook*.

Identifier	Data Type	Changeable?	Description
"#AUTO GROW FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — sort up, no duplicates, click/release, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#AUTO GROW HORIZONTAL"	Bit	Yes	-1 if slave (horizontal) is enabled, 0 if disabled.
"#AUTO GROW VERTICAL"	Bit	Yes	-1 if slave (vertical) is enabled, 0 if disabled.
"#NO AUTO GROW"	Bit	Yes	-1 if don't adjust form is enabled, 0 if disabled.
"#AUTO GROW ICON"	Bit	Yes	-1 if draw grow icon is enabled, 0 if disabled.
"#AUTOGROW PROCEDURE"	Bit	Yes	-1 if .Autogrow proc is enabled, 0 if disabled.

Super Matrix SuperObject™ Commands

The Super Matrix SuperObject (see “[Super Matrix Objects](#)” on page 939 of the *Panorama Handbook*) understands a small set of commands that can be sent to it with the `superobject` statement in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). This table describes each of these commands in detail.

Command	Parameters	Description
"ReDraw"	Area,Start,End	<p>This command redraws some or all of the cells in a super matrix. The first parameter, Area, defines the area that will be redrawn. Legal options for this parameter are: "all", "column", "row", and "cell".</p> <p>The Start and End parameters define the start and end of the area to be redrawn. For example, if the Area parameter was "column" and the last two parameters were 3 and 5, then columns 3 thru 5 would be redrawn. (Note: The start and end values are ignored if the "all" area is chosen.)</p> <p>The following examples illustrate different ways a matrix might be updated. This calendar example redisplay the entire month.</p> <pre>superobject "Month","redraw","all",0,0</pre> <p>This example redisplay only weekdays.</p> <pre>superobject "Month","redraw","column",2,6</pre> <p>This example works with a matrix of photographs. The procedure redisplay photo 7 only.</p> <pre>superobject "Photographs","redraw","cell",7,7</pre> <p>This example redisplay all photos after photo 12. This procedure would be used if someone inserts or deletes a photograph at position 12.</p> <pre>superobject "Thumbnails","redraw","cell",12,9999</pre>
"CellRectangle"	Cell,Rectangle	<p>This command returns the dimensions of an individual cell in the matrix. The dimensions are in window co-ordinates. The Cell parameter should be a number from 1 to the maximum number of cells in the matrix. The Rectangle parameter should be a field or variable where the rectangle will be stored.</p> <p>This example uses the "CellRectangle" command to open a new window over the current matrix cell (the matrix cell that was clicked on).</p> <pre>local subWindowRectangle superobject "Calendar","CellRectangle", info("matrixcell"),subWindowRectangle setwindowrectangle xytoxy(subWindowRectangle,"w","g"),"" openform "Day"</pre>
"CellToXY"	Cell,Row,Col	<p>This command converts a matrix cell number into a row and column. The example below displays what row and column were clicked on.</p> <pre>local mRow,mCol superobject "Thumbnail", "CellToXY",info("matrixcell"),mRow,mCol message "You clicked on row "+str(mRow)+ " and column "+str(mCol)</pre>

Command	Parameters	Description
"XYToXY"	Cell,Rectangle	This command converts a rectangle within the matrix frame object into a rectangle within an individual cell in the matrix. The intended use for this command is allow a procedure to test whether the mouse is over a specific object within the cell, or to pop-up a text editor over a specific section of the cell. The Cell parameter should be a number from 1 to the maximum number of cells in the matrix. The Rectangle parameter should be a rectangle inside the frame rectangle. For an example of this command in action see the <code>.matrixClick</code> procedure in the Albums example database.
"MatrixShape"	Row,Column	This command changes the shape of the matrix. The parameters specify the visible rows and columns. For example, if you want to display 3 by 5 matrix the parameters should be 3,5. If negative, these are fixed heights/widths (in pixels). For example, to display 64 by 64 icons use -64,-64.
"MatrixGetShape"	Row,Column	This command returns the current shape of the matrix. Positive numbers indicate the number of rows or columns, negative numbers indicate a fixed height or width in pixels. For example -20,1 indicates that there is one column, with a variable number of rows that are each 20 pixels high.
"MatrixScroll"	Row,Column	This command moves the upper left hand corner of the matrix to the specified coordinates, adjusting the scroll bars as necessary. The coordinates are number from 1..N. If there is only one scroll bar then the other parameter is ignored.
"MatrixGetScroll"	Row,Column	Get the coordinates of the upper left hand cell, starting from 1. As you scroll down and to the right, the numbers will increase.
"PointToCell"	Point,Cell	Given a point in local co-ordinates, it returns the corresponding cell # (or zero if not in a cell).
"MatrixBounds"	Cells/Rows,Columns	This command sets the bounds of the scrollable area. If there is only one scroll bar, then only the first parameter is used and it is treated as the number of CELLS in the matrix. For example, if your matrix is displaying 97 items then you would set the number of cells to 97. This means that when you drag the scroll thumb to the bottom you'll see the last row of items, up to number 97. If there are two scroll bars then the parameters are rows and columns. Until this statement is used the default is 100 cells or rows/columns. (Note: If you have specified the matrix data formula and separator then the matrix bounds will be calculated automatically, and you should not use this command.)
"MatrixSize"	Cells	This command calculates the current number of cells in the matrix. Here is a procedure that displays all the vital statistics for a matrix. <pre> local mCells,mRows,mCols superobject "Images","MatrixSize",mCells superobject "Images","CellToXY",mCells,mRows,mCols message "This matrix contains "+ str(mCells)+" cells ("+ str(mRows)+" rows by "+ str(mCols)+" columns). </pre>
"Scroll"	Rows,Cols	This command slides the display of the matrix up or down and/or right or left. This command simply slides the matrix display — it's up to you to adjust the underlying data structure. If the Rows parameter is positive the matrix display will slide up by the specified number of rows, if negative it will slide down. If the Cols parameter is positive the matrix display will slide right by the specified number of columns, if negative it will slide left. For either parameter a value of 0 may be used to maintain the same position on that axis. Note: This command is obsolete, it was designed before matrix objects had scroll bars.

Super Matrix SuperObject Internal Data

This table describes the internal data in a Super Matrix SuperObject that can be accessed and modified using the “back door” described in “[Internal Data Types](#)” on page 669. To learn more about how these options work see “[Designing a Matrix Template](#)” on page 949 of the *Panorama Handbook*.

Identifier	Data Type	Changeable?	Description
"#SUPER MATRIX FLAGS"	Long Word	Yes	This internal data item contains all of the on/off options for the object — matrix order, fixed width, fixed height, click/release, etc. You can also access each of these options separately (see following entries). Being able to access all of these values at once makes it easy to save all the flags, modify selected flags, and then restore all of the original settings.
"#VERTICAL SCROLL BAR"	Bit	Yes	-1 if vertical scroll bar is enabled, 0 if disabled.
"#HORIZONTAL SCROLL BAR"	Bit	Yes	-1 if horizontal scroll bar is enabled, 0 if disabled.
"#THIN SCROLL BARS"	Bit	Yes	-1 if thin scroll bars are enabled, 0 if disabled.
"#GROW BOX"	Bit	Yes	-1 if grow box is enabled, 0 if disabled.
"#SUPER MATRIX SHOW FRAME"	Bit	Yes	-1 if display:frame object is enabled, 0 if disabled.
"#SUPER MATRIX ORDER"	Bit	Yes	-1 if horizontal is enabled, 0 if vertical .
"#SUPER MATRIX FIXED WIDTH"	Bit	Yes	-1 if fixed width (pixels) is enabled, 0 if fixed # of columns .
"#SUPER MATRIX FIXED HEIGHT"	Bit	Yes	-1 if fixed height (pixels) is enabled, 0 if fixed # of rows .
"#CLICK/RELEASE"	Bit	Yes	-1 if click/release is enabled, 0 if disabled.
"#SUPER MATRIX GROW METHOD"	Bit	Yes	-1 if slide is enabled, 0 if proportional .
"#SUPER MATRIX BORDERS"	Bit	Yes	-1 if cell borders is enabled, 0 if disabled.
"#OVERFLOW"	Bit	Yes	-1 if overflow is enabled, 0 if disabled.
"#SUPER MATRIX COLUMNS"	Long Word	Yes	Number of columns if fixed # of columns is enabled, or width of each column (in pixels) if fixed width is enabled.
"#SUPER MATRIX ROWS"	Long Word	Yes	Number of rows if fixed # of rows is enabled, or width of each column (in pixels) if fixed height is enabled.
"#SUPER MATRIX GROW BOUNDARY"	Long Word	Yes	Point specifying slide boundaries (used if slide is enabled). Use v() and h() functions to extract individual dimensions (see “ Points ” on page 147).
"#SUPER MATRIX FRAME"	Text	Yes	Name of matrix frame object (maximum 31 characters).
"#PROCEDURE"	Text	Yes	Name of procedure triggered when matrix is clicked on, if any.
"#SEPARATOR"	Byte	Yes	Array separator character, if any.
"#FORMULA"	Text	Yes	Formula for data array, if any.

Scroll Bar SuperObject™ Commands

The Scroll Bar SuperObject (see “[Scroll Bars](#)” on page 979 of the *Panorama Handbook*) understands a small set of commands that can be sent to it with the `superobject` statement in a procedure (see “[Program Control of SuperObjects™](#)” on page 666). This table describes each of these commands in detail.

Command	Parameters	Description
"GetScrollMin"	Value	This command gets the minimum scroll bar value and places into the field or variable specified by Value .
"SetScrollMin"	Value	This command sets the minimum scroll bar value to any numeric value (must be integer) between 1 and 65535. (This value is normally set by the Min value in the Scroll Bar dialog.)
"GetScrollMax"	Value	This command gets the maximum scroll bar value and places into the field or variable specified by Value .
"SetScrollMax"	Value	This command sets the maximum scroll bar value to any numeric value (must be integer) between 1 and 65535. (This value is normally set by the Max value in the Scroll Bar dialog.) Here is an example that sets the maximum value of the scroll bar named Slider to the number of elements in the array People : <pre>superobject "Slider", "SetScrollMax", arraysize(People, 1)</pre>
"GetScrollPage"	Value	This command gets the scroll bar page amount and places into the field or variable specified by Value . This value is the amount the scroll bar value will increase or decrease if the user clicks on the gray area above or below the thumb of the scroll bar.
"SetScrollPage"	Value	This command sets the scroll bar page amount to any numeric value (must be integer) between 1 and 65535. This value is the amount the scroll bar value will increase or decrease if the user clicks on the gray area above or below the thumb of the scroll bar. (This value is normally set by the Page Up/Down value in the Scroll Bar dialog.)
"GetScrollValue"	Value	This command gets the current position of the scroll bar. This command is redundant because you can always get the position by examining the field or variable linked to the scroll bar.
"DisableScroll"		This command disables the scroll bar. The scroll bar is still visible, but it turns white and the thumb disappears.
"EnableScroll"		This command enables the scroll bar (see DisableScroll above).
"GetScrollEnable"	Value	This command checks to see if a scroll bar is enabled or disabled, and sets the field or variable specified by Value with a true or false result accordingly.

Speech Synthesis

Speech synthesis is an exciting new feature introduced in Panorama V. (Note: Speech synthesis is available only on Macintosh systems. It is not available on Windows systems.) There are several new statements and functions that support speech synthesis. In its simplest form you can cause Panorama to speak a word or phrase simply by using the `speak` statement followed by the text to speak.

The Speak Statement

This statement speaks a word or phrase using the current voice. If the voice is already talking, it waits until the current text has been spoken before speaking the new text. The program continues running immediately - in other words, the next statement will be executed immediately, without waiting for the text to be spoken. See the `info("speaking")` function if you need to wait until the text has been spoken to proceed.

The `speak` statement has one parameter, the text to be spoken. Here is an example that speaks a short sentence.

```
speak "Welcome to the order entry system!"
```

Embedded Speech Commands

The system does its best to parse the text you supply and to speak it properly. For example, it will automatically inflect the end of a sentence depending on whether the sentence ends with a period, question mark, or exclamation mark. You can also tweak the way the text is pronounced with embedded speech commands. These commands are text tags that are embedded in the text to change the rate, pitch and emphasis of the text being spoken. To learn about how to use these tags see *Embedded Speech Commands* on Apple's developer website.

```
http://developer.apple.com/techpubs/mac/Sound/Sound-200.html#HEADING200-0
```

The StopSpeaking Statement

This statement tells Panorama to shut up now! Any text that is being spoken will stop immediately in mid-sentence. There are no parameters.

The info("speaking") Function

This function returns true if talking is occurring right now. You can use this function to wait for speech to finish before continuing to the next step in a procedure. For example, this procedure times now long it takes to speak a phrase.

```
local start
start=now()
speak "How long does it take to say this?"
loop
  nop
while info("speaking")
  message now()-start
```

Buffered Speech

The buffered speech statements allow you to build up the text you want to speak item by item. For example, you might want to include data from different records in the text to be spoken. The buffered speech statements allow you to build up the text to be spoken over several statements, then smoothly speak the combined text when you are done assembling the items. The two basic buffered speech statements are `speakwords` and `speaknow`. Here's a silly example of how these could be used to count from one to five.

```
speakwords "One"
speakwords "Two"
speakwords "Three"
speakwords "Four"
speakwords "Five"
speaknow
```

Before moving to a less silly example, the table below describes each of the statements for buffering speech. As the table shows, each of these statements can speak text, numbers, dates, or some combination.

Data Type	Statement	Description
TEXT	<code>speakwords</code>	The text is spoken using normal English.
	<code>speakdigits</code>	The data is spoken in English, but if the data contains any numbers they will be spoken as individual digits. For example, <code>4892</code> will be spoken as <code>four eight nine two</code> .
	<code>speakdigitpairs</code>	The data is spoken in English, but if there are any numbers they will be spoken as digit pairs. For example, <code>4892</code> will be spoken as <code>forty-eight ninety-two</code> .
	<code>speakletters</code>	The data is spoken letter by letter, including upper and lower case. For example <code>Jim</code> will be spoken as <code>upper case J, lower case I, lower case M</code> .
	<code>speakcharacters</code>	The data is spoken letter by letter. For example <code>April</code> will be spoken as <code>A P R I L</code> .
	<code>speakcharactersslowly</code>	The data is spoken letter by letter, with a delay between each letter. For example <code>April</code> will be spoken as <code>A; P; R; I; L</code> .
	<code>speakaddress</code>	The data is spoken in English, but any official USPS abbreviations are expanded as the text is spoken. For example <code>S</code> or <code>S.</code> is expanded to <code>South</code> , while <code>Bl.</code> is expanded to <code>Boulevard</code> . Any numbers in the text are spoken as digit pairs.
	<code>speakstate</code>	If the text is a two letter abbreviation of a U.S. state it will be spoken as the full name. For example <code>CA</code> will be spoken as <code>California</code> , while <code>WV</code> will be spoken as <code>West Virginia</code> .
	<code>speakphonenumber</code>	The first ten digits in the text are spoken as a phone number, along with any extension after that.
NUMBER	<code>speakwords</code>	The number is spelled out. For example, <code>4892</code> will be spoken as <code>four-thousand eight-hundred and ninety-two</code> .
	<code>speakdigits</code>	The number is spoken as individual digits. For example, <code>4892</code> will be spoken as <code>four eight nine two</code> .
	<code>speakdigitpairs</code>	The number is spoken as digit pairs. For example, <code>4892</code> will be spoken as <code>forty-eight ninety-two</code> .
	<code>speakdollars</code>	The integer portion of the number is spelled out, followed by the word Dollars. For example <code>378.93</code> will be spoken as <code>Three-hundred seventy-eight dollars</code> .
	<code>speakdollarsandcents</code>	The integer portion of the number is spelled out, followed by the word Dollars, followed by the two digits after the decimal point, followed by the word Cents. For example <code>378.93</code> will be spoken as <code>Three-hundred seventy-eight dollars and ninety-three cents</code> .
DATE	<code>speakdate</code>	The date is spelled out. For example <code>4/18/04</code> is spoken as <code>April eighteenth, 2004</code> .

Here is an example that reads a check from a checkbook database:

```

speakwords "Check number"
speakdigitpairs Check
speakwords "to"
speakwords «Pay To»
speakwords "for"
speakdollarsandcents Debit
speaknow

```

This example reads an address from a contact database.

```

speakwords First+" "+Last
speakaddress Address
speakwords City+", "
speakstate State
speakdigits Zip
speaknow

```

Speaking Using the Speech Wizard

The **Speech Wizard** (see “[Speech Wizard](#)” on page 103 of *Wizards & Demos*) allows you to define templates for speaking the data in each database. To speak the data in the current record using the currently active template use the **speakthisrecord** command (no parameters are required.)

To speak the data in the current record using a specific template use the **speakscript** statement.

```
speakscript database,template
```

If the **database** is "" the current database will be assumed, otherwise you can specify any open database.

The **template** you specify must have been previously defined in the speech wizard. You can use the **speechscripts** command to find out what templates are available.

```
speechscripts database,templatelist
```

If the **database** is "" the current database will be assumed, otherwise you can specify any open database.

The **templatelist** must be the name of a field or variable. When the statement is complete, this field or variable will contain the names of the templates that have been created in the Speech Wizard, one template per line.

The example below will speak the current record over and over again using each template that has been set up in the Speech Wizard.

```

local tlist,tp,n
speechscripts "",tlist
n=1
loop
    tp=array(tlist,n,¶)
    stoploopif tp=""
    speakscript "",tp
    n=n+1
while forever

```

Printing

Even in this e-commerce age many jobs still require paper output. Printing can be done manually (see “[Printing Basics](#)” on page 1047 of the *Panorama Handbook*) or via a procedure.

Selecting a View for Printing

In Panorama printing is always done through a specific view. You can always print the data sheet, but usually when printing with a procedure you’ll be using a form set up with a custom report (see “[Custom Reports](#)” on page 1061 of the *Panorama Handbook*). Before printing begins the procedure must select the appropriate form, either in a new window with the `openform` statement (see “[Opening a Window](#)” on page 445) or within the current window with the `goform` statement (see “[Changing a Window’s View](#)” on page 451). An alternate way to select a view for printing is to use the `printusingform` statement. This statement allows a procedure to print using a form without actually opening the form. See “[Printing Using an Alternate Form](#)” on page 725 to learn how to use this statement.

Selecting a Printer

A procedure always prints to the currently selected printer. The currently selected printer can be changed manually using your system software, or if you are using OS X you can also change the printer via a procedure.

Changing the Current Printer

To change the current printer use the `changeprinter` statement:

```
changeprinter printername
```

The printer name must be the name of one of the printers connected to the system (you can use the `listprinters()` function for a list, see below). Here is a procedure that opens a form and prints a check on an Epson printer.

```
openform "CheckTemplate"
changeprinter "Epson Stylus CX5400"
printonerecord dialog
closeform
```

Note: If the current window is a form with a default printer (see “[Setting up Default Printers](#)” on page 1059) the `changeprinter` statement is ignored.

Changing the Default Printer

The `changedefaultprinter` statement will change the default printer for one or more forms (see “[Setting up Default Printers](#)” on page 1059).

```
changedefaultprinter printername,forms
```

The second parameter, `forms`, is a carriage return delimited list of forms. This procedure will set the default printer of both the `Checks` and the `Deposits` forms to use the Epson printer.

```
changedefaultprinter "Epson Stylus CX5400", "Checks"+cr()+"Deposits"
```

This procedure will set the default printer of all forms in the current database to the Epson printer.

```
changedefaultprinter "Epson Stylus CX5400",dbinfo("forms","")
```

If no form is specified then the current form is assumed.

```
changedefaultprinter "Epson Stylus CX5400", ""
```

If no printer is specified then the default printer is removed from the form (or forms). This procedure removes the default printer from all forms in the current database.

```
changedefaultprinter "",dbinfo("forms","")
```

After this procedure is used all of the forms will print using the currently selected printer.

Getting Information About Printers

Use the `listprinters()` function to get a list of printers that are available on this computer. Depending on what printers are available this will return a carriage return delimited array something like this:

```
HP Laserjet 5P
LaserWriter 12/640 PS
Stylus CX5400
```

Just because a printer is on this list does not mean that it is physically able to print at this time. Unfortunately there is no reliable method for a procedure to determine if a printer is actually currently available for printing. You can use the `printerstatus(printername)` function to check printer status, but it may show `stopped` or `idle` whether a printer is actually available or not.

Use the `currentprinter()` function to find out what the currently selected printer is.

Adjusting Page Setup

The **Page Setup** dialog allows you to configure various printing options (see “[The Page Setup Dialog](#)” on page 1053). A procedure cannot control these options directly, but it can open the **Page Setup** dialog automatically using the `pagesetup` statement. This statement does not have any parameters, it is simply used by itself. Here is a simple procedure that opens a form and allows the page setup to be adjusted, then closes the form.

```
openform "My Report"
pagesetup
closewindow
```

Panorama keeps a separate **Page Setup** configuration for each form. This allows different forms to have different configurations (for example portrait vs. landscape orientation).

Preparing Data For Printing

Most procedures that print the database also prepare the database in some way. Typically, a procedure may sort, select a subset of the database and/or prepare summaries. See “[Sorting](#)” on page 551, “[Locating Information](#)” on page 552 and “[Summaries and Outlines](#)” on page 566 to learn how to perform these tasks with a procedure.

Printing the Database

A procedure can print all selected records in the current database using the `print` statement. This statement may be used one of two ways. The first method is to follow the statement with the parameter `dialog`, like this (there must be a space between `print` and `dialog`, as shown below).

```
print dialog
```

When used this way the procedure will pause and display the standard **Print** dialog, the same dialog that normally appears when you choose the **Print** command. This allows you to choose the print options for this print run (number of copies, paper source, etc.) The exact options depend on the printer you have selected. When the **Print** button is pressed the procedure will go ahead and print all the selected records in the current database.

The second method is to follow the `print` statement with the parameter `" "`, as shown here.

```
print " "
```

When the print statement is used this way it will not display the **Print** dialog. Instead, it simply prints the database using the same options that were used the last time this form was printed. (Note: Depending on the printer driver software your printer uses, some options may not be saved from print to print. These options will use default settings. For absolute control over all print options we recommend that you use the `dialog` option.)

Here is an example of a complete procedure to print a report. The procedure opens the form that is designed for printing this report and then selects the appropriate data. After printing it selects all of the data again and then closes the form.

```
openform "90 Day Report"
select Date>today()-90
print dialog
selectall
closewindow
```

Printing a Single Record

To print just the current record use the `printonerecord` statement. For example this statement could be used to print a single letter or a single invoice. Like the `print` statement the `printonerecord` statement may be used with a parameter of either `dialog` or `" "`. Here is an example that prints the current invoice.

```
openform "Paper Invoice"
printonerecord dialog
closewindow
```

Print Preview

The `printpreview` statement opens a special preview window. This window displays a preview of the printed results for the current view. This is the same as choosing **Preview** from the File menu (see "[Print Preview](#)" on page 1056 of the *Panorama Handbook*). The user can flip forward to see additional pages of the previewed report. When the user closes the preview window, the procedure continues with the statement after the `printpreview` statement. Usually this should be either the end of the procedure or the `stop` statement (see "[Stopping the Program](#)" on page 278).

The most common reason to use the `printpreview` statement in a procedure is to simulate the **Print Preview** command in your own custom File menu. Here are the statements to use in your `.CustomMenu` procedure (see "[The .CustomMenu Procedure](#)" on page 363). (You could also trigger print preview with a button.)

```
if info("trigger") beginswith "Menu.File.Print Preview"
    printpreview
    stop
endif
```

Printing Using an Alternate Form

The `printusingform` statement allows the current database to be printed using a different form than the one currently being displayed. It is designed to be used in combination with the `print`, `printonerecord`, or `printpreview` statements (see previous sections). This statement has two parameters: `file` and `form`.

```
printusingform file,form
```

File is the name of the database file that contains the form to be printed. The database file must be open. Usually the form will be in the current database, and in that case you can simply use an empty string ("") for the file name. **Form** is the name of the form to be printed.

The `print` statement normally prints whatever window is currently active. If you want to print a different window, you must first open that window and then print (see “[Selecting a View for Printing](#)” on page 722). The `printusingform` statement is another way to print an alternate form.

Warning: The `printusingform` statement may only be used when a form window is currently on top. It will not work when a data sheet window is the current window.

The procedure below will print **My Report**, even if another form is currently visible.

```
printusingform "", "My Report"  
print dialog
```

The procedure below will print **Standard Report #4** from the **Reports** database. Although the form is from the **Reports** database, the data will be from the current database. This usually only makes sense if the two databases have the same fields.

```
printusingform "Reports", "Standard Report #4"  
print dialog
```

Printing Data in an Array

The `printonemultiple` statement prints a form over and over again without advancing from record to record. Instead of advancing from record to record, a variable is incremented each time the form is printed. This statement is designed for printing information in an array (see [“Text Arrays”](#) on page 93) using a Super Matrix (see [“Super Matrix Objects”](#) on page 939 of the *Panorama Handbook*). Typical examples include calendars and photo thumbnails. The `printonemultiple` statement has five parameters.

```
printonemultiple variable,start,end,bump,copies
```

The `variable` parameter is the name of the variable you wish to increment as each page is printed.

The `start` parameter is the beginning sequence number or date value. `Start` can be an integer number, a variable containing a numeric integer or date value, or a formula or function which results in a numeric integer or date value. The `start` parameter must be less than or equal to the `end` parameter.

The `end` parameter is the ending sequence number or date value. `End` can be an integer number, a variable containing a numeric integer or date value, or a formula or function which results in a numeric integer or date value. `End` must be greater than or equal to `start`.

The `bump` parameter is the increment value for your sequence. `Bump` may be a number, a numeric variable, or a formula which results in a positive numeric integer. The `bump` value must be a positive integer for a numeric field. For a date field `bump` may also be one of the following:

Bump	Description
"M"	bump one month per page
"Y"	bump one year per page
1	bump one day per page
7	bump one week per page

The `copies` parameter is the number of times the form is printed for each sequence number. `Copies` may be a number, a numeric variable, or a formula which results in a positive numeric integer (usually 1).

The `printonemultiple` statement will print a form a predetermined number of times. Each printing of the form may be sequenced with incrementing integer or date values in a specified variable. Note: the `printonemultiple` statement does not actually print itself, but must be followed by a `printonerecord` statement (see [“Printing a Single Record”](#) on page 724).

This example prints the next 3 months of a monthly calendar. The example assumes that the form `Monthly Calendar` will display the month specified by the variable `CalendarDate`.

```
fileglobal CalendarDate
openform "Monthly Calendar"
printonemultiple CalendarDate,today(),today()+90,"m",1
printonerecord dialog
closewindow
```

This example prints all the photograph files in the current folder. The example assumes that the form `Picture Matrix` will display 20 pictures per page, probably using a SuperMatrix object. (It's possible for a procedure to find out how many pictures are on each page, see [“Super Matrix SuperObject™ Commands”](#) on page 715). The picture in the top left corner of each is controlled by the global variable `PicNumber`.

```
fileglobal PicNumber,PicMax
PicMax=arraysize( listfiles("", "PICT"),1)
openform "Picture Matrix"
printonemultiple PicNumber,1,PicMax,20,1
printonerecord dialog
closewindow
```

Printing Directly to a PDF File

If you are using Mac OS X and have the **CodePoetry CUPS-PDF Package** installed (see “[Installing the CUPS-PDF Package](#)” on page 728) your procedures can print directly to a PDF file with no user intervention (no dialogs appear, etc.). This is done with the `printpdf` statement.

```
printpdf options
```

If the `options` parameter is left blank this statement will simply “print” a PDF file from the current database view. The PDF file will be given the same file as the database with `.pdf` added to the end of the name.

For example, suppose this statement appears in a procedure in a database named **Contacts**:

```
printpdf ""
```

This will create a file named **Contacts.pdf** in the same folder as the database.

The options parameter can contain one or more “assignments” to modify the operation of the `printpdf` statement. Each assignment takes the form `option=value`, for example `{file="Invoice 3412.pdf" onerecord=true}`. There are five different options that can be specified.

Option	Description
<code>file=</code>	Specifies the name of the PDF file to be generated. If "" or if this option is omitted the name of the database is used (with <code>.pdf</code> appended)
<code>path=</code>	Specifies the location of the PDF file to be generated. If "" or if this option is omitted the PDF file is created in the same folder as the database.
<code>form=</code>	Specifies the form to be printed. If "" or if this option is omitted the current view is printed. The specified form does not have to be open, but the current window must be a form window.
<code>database=</code>	Specifies the database to be printed (the specified database must be open). If "" or if this option is omitted current database is printed. If an alternate database is specified you must also specify a form with the <code>form=</code> option.
<code>onerecord=</code>	May be either true or false. If true only the current records will be printed. If false (or if "") then all selected records will be printed.

Here is an example for printing the current invoice from an invoice database. The PDF file will be saved into a folder named **Printed** that is a subfolder of the folder that contains the invoice database.

```
printpdf {file="Invoice "+str(«InvoiceNumber»)+".pdf" }+
  {path="+subpath(dbfolder(),"Printed")+"}"+
  {onerecord=true}
```

This example prints the contents (all selected records) of the **Checkbook** database into a file named **Reports.pdf** on the desktop. The data will be formatted using the **Transactions** form.

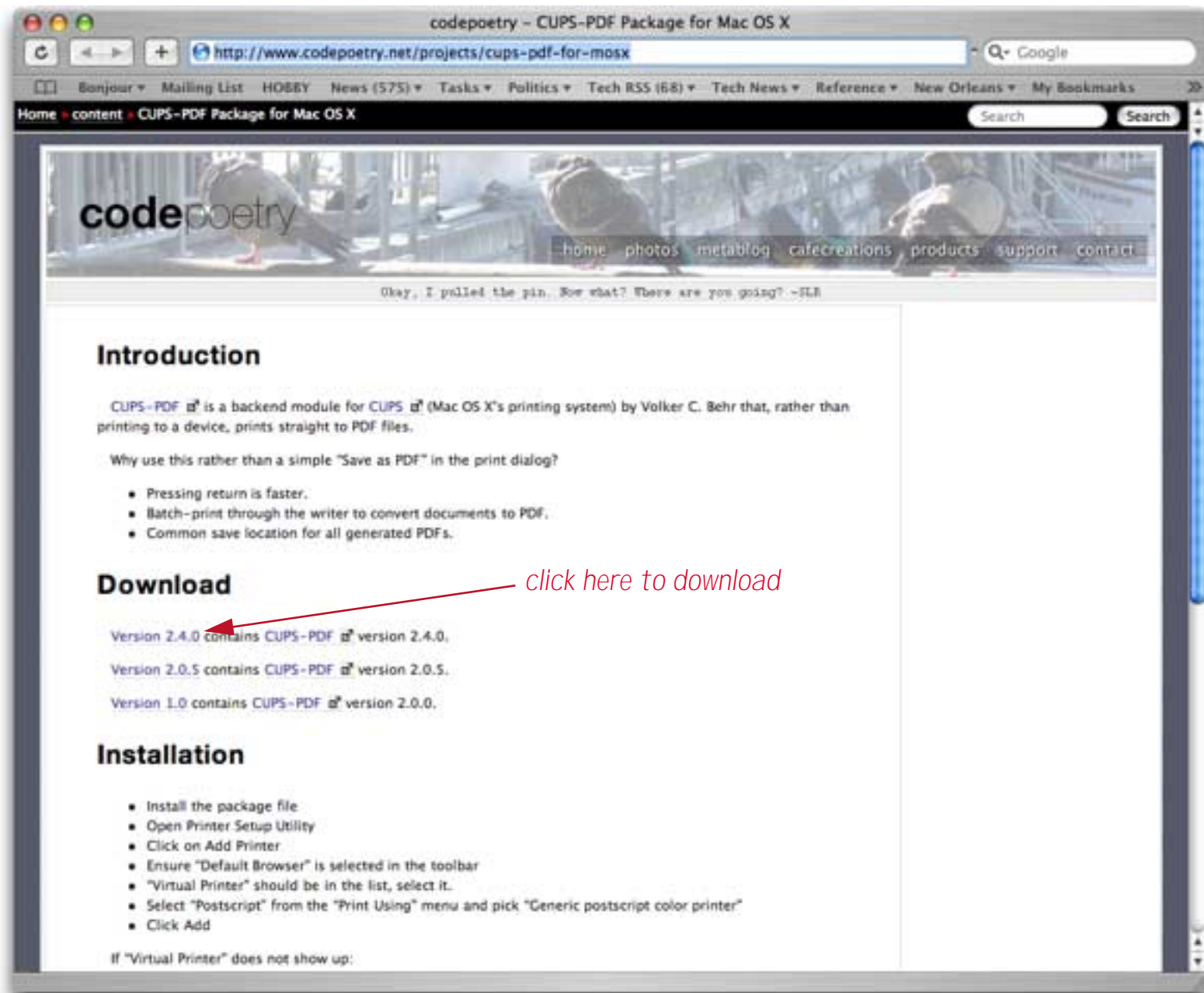
```
printpdf {file="Report.pdf" path="+folderpath(info("desktopfolder"))+" }+
  {database="Checkbook" form="Transactions" }
```

Installing the CUPS-PDF Package

Before you can use the `printpdf` statement you must download and install the **CodePoetry CUPS-PDF Package**. CUPS-PDF is an open source program for creating PDF files that can be freely downloaded. The OS x version of this package can be downloaded from the CodePoetry web site.

<http://www.codepoetry.net/projects/cups-pdf-for-mosx>

At the time this was written the CodePoetry web site looked like this:



The first step is to download the installer. Once this is complete the installer should automatically launch.



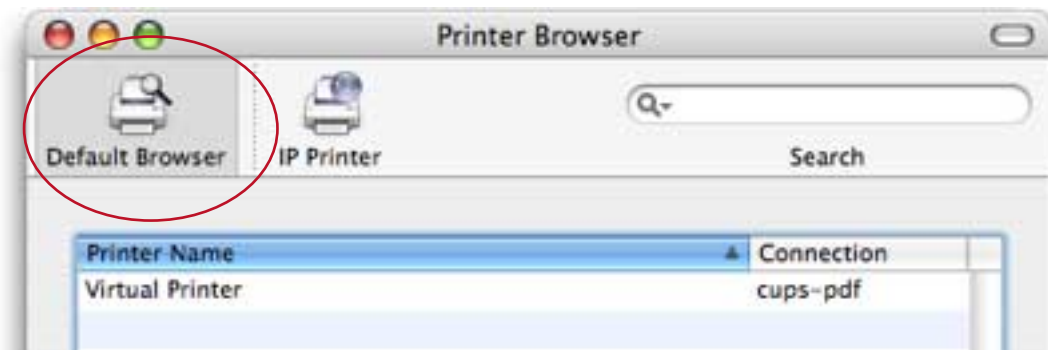
Follow the steps until the package is installed. After the package is installed open the **Printer Setup Utility**. This is usually found in the Utilities folder inside the Applications folder.



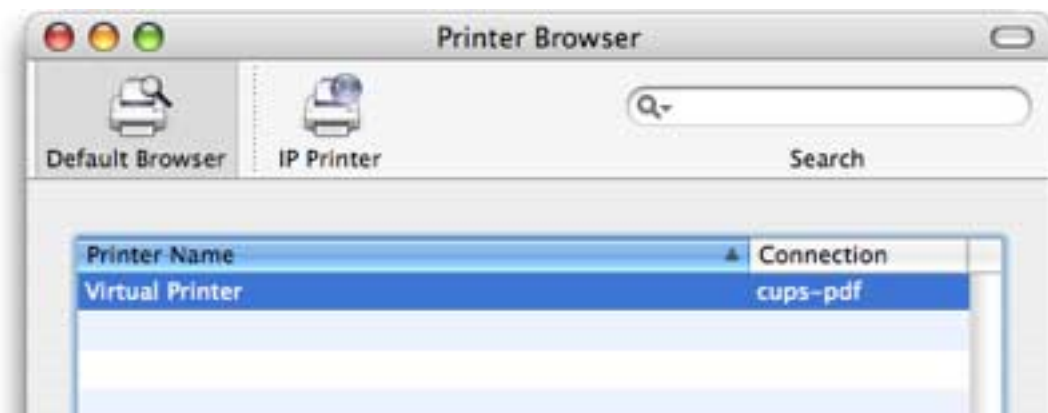
Next, click on the **Add** button or choose Add Printer from the Printers menu.



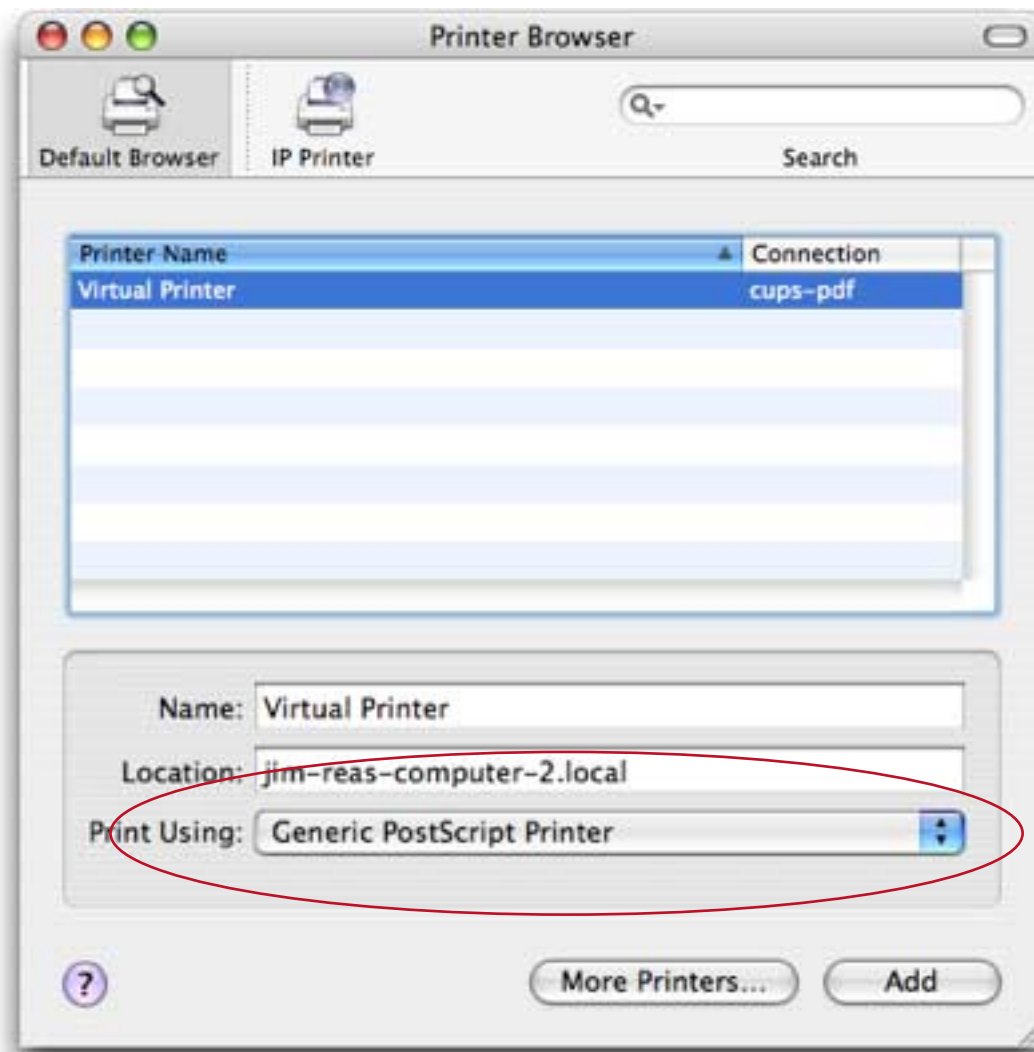
Now make sure that **Default Browser** is selected in the toolbar.



The next step is to select **Virtual Printer** from the list of printers.



Select Generic postscript color printer from the Print Using pop-up menu.



To finish press the **Add** button and exit from the **Printer Setup Utility**. Your system is now ready to use the `printpdf` statement (see “[Printing Directly to a PDF File](#)” on page 727).

Form Comments

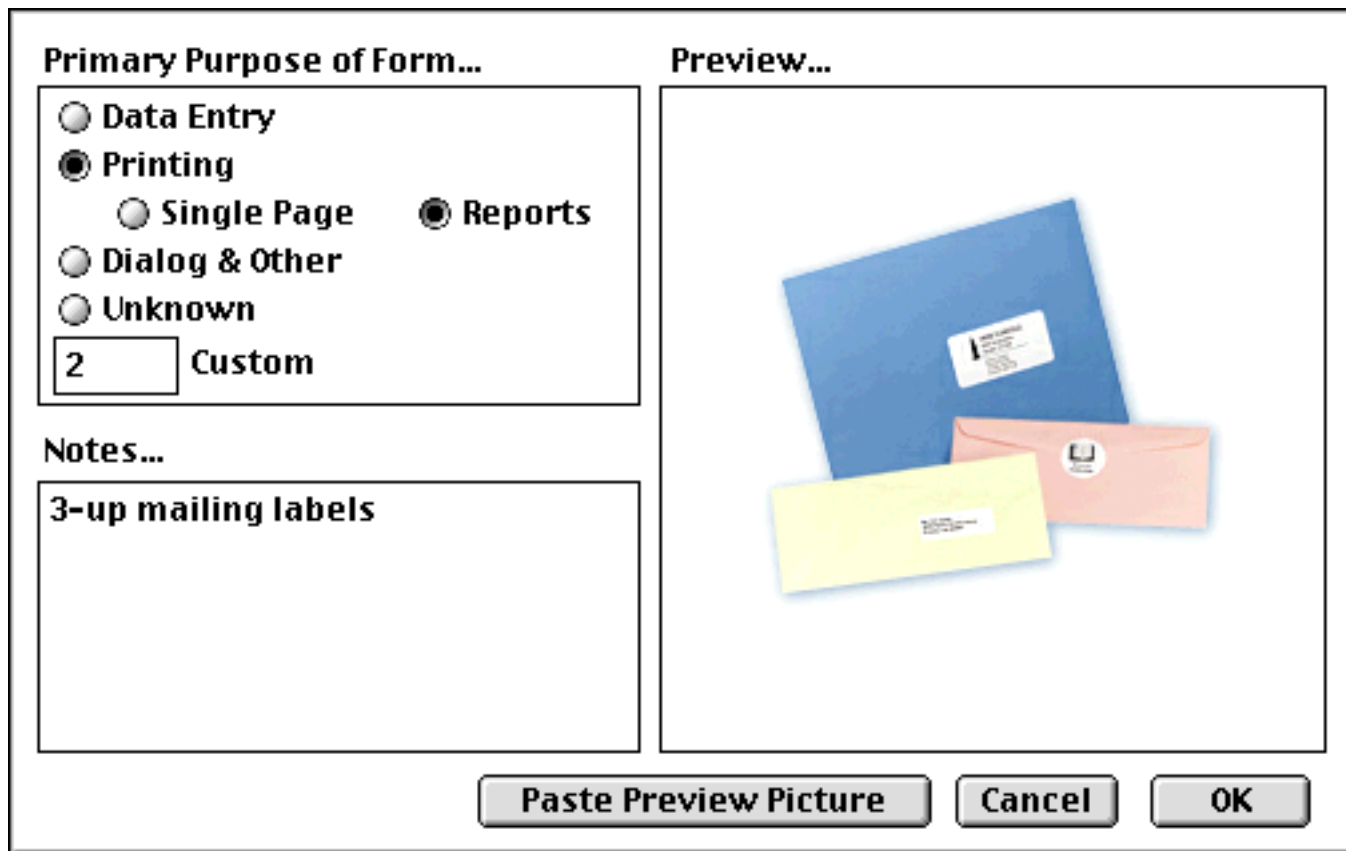
Panorama allows you to create extra explanation comments for each form in a database. These comments let you keep some notes to yourself about each form—what its purpose is, what kind of paper it is printed on, whatever you want to remember. There's a limited amount of space, however, so don't go into great detail about each field in the form. To create these comments, use the **Form Comments** command in the Setup Menu (graphics mode only). Enter the comments in the box in the lower left hand corner of the dialog.

You can also assign a form type using the radio buttons in the upper left hand corner of the dialog. This type is for your information only and may also be used to select classes of forms using the `formselect` procedure statement. The pre-defined form types are:

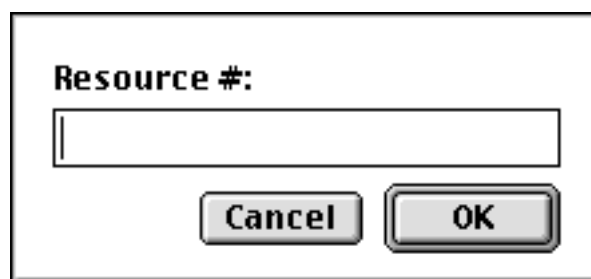
Option	Description
Data Entry	This is for forms that are primarily for data entry.
Printing	This is for forms that are primarily designed for printing. This option is further subdivided into single page forms that are designed for printing individual records, for example checks or tax forms, and reports that are designed for printing many records at a time.
Dialog & Other	This is for forms that are designed to be displayed as dialog boxes.
Unknown	This is the default setting before a purpose has been assigned.
Custom	Allows you to create your own form classifications (for use with the <code>formselect</code> statement - see " The FormSelect Statement " on page 734). Custom classes may be numbers from 10 to 255.

Remember, these form types are for your information only—Panorama will not stop you from printing a form that is designated as a dialog or from editing data in a form that is designated as a report. However, form types can be very useful to help the database designer (this means you) keep track of what forms are for what.

In addition to the text comments, you can also assign a preview picture to the form. Before opening the **Form Comments** dialog, copy the picture into the clipboard. Once the dialog is open, you can use the **Paste Preview Picture** button to paste the picture into the comment window.



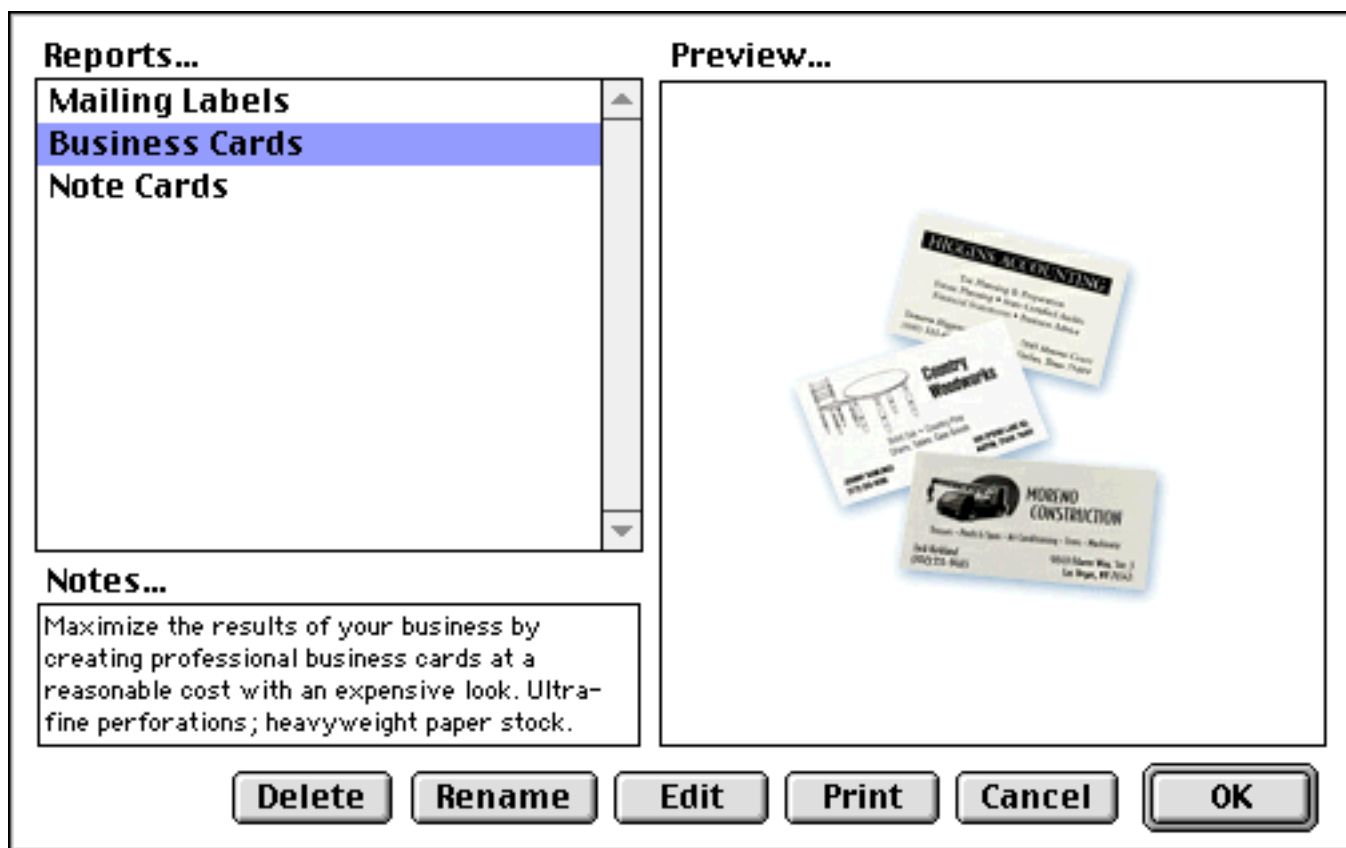
Another way to paste pictures into a form comment is to put the pictures in a resource file. The advantage of this approach is that the picture doesn't waste any of your valuable memory. However, the resource file must be opened in the **.Initialize** procedure for this to work (see "[Opening and Closing Resource Files](#)" on page 435). Once the picture is stored in the resource file (see "[Working with Resources](#)" on page 433), you can open the **Form Comments** dialog, then hold down the **Option** key and press the **Paste Preview Picture** button. Panorama will request the number of the resource containing the picture. Enter the number and press **Enter**.



No matter how you get the picture into Panorama, the picture itself should be no more than 256 pixels high by 256 pixels wide.

The FormSelect Statement

The `formselect` statement pauses a procedure and displays a dialog through which the user can choose a form from the active database. The dialog may also show the Form Comments information (see “[Form Comments](#)” on page 732.) The dialog will look something like this.



The `formselect` statement has four required parameters.

```
formselect dialog,filter,button,form
```

Dialog is the resource number that identifies the dialog you wish to display. If you do not wish to create your own dialog, with ResEdit for example, you may use Panorama's built in dialog number **2086**. **Filter** is a numeric value used to determine which type of forms will be displayed in the dialog (see “[Form Comments](#)” on page 732). The following table shows the possible filter values.

Value	Selected Forms
0	All forms
1	Data Entry forms
2	Printing forms
3	Dialog & related forms
4 or greater	Custom forms

Button is the name of a variable that will contain the name of the button that was pressed inside the `formselect` dialog. Clicking on any button in the dialog closes the dialog and allows the procedure to continue.

Form is the name of a variable that will contain the name of the form selected in the dialog. If the variable is pre-set to the form name before the `formselect` statement is reached this form will be selected when the dialog opens. If no form is selected this variable will equal "" .

This example opens the built-in Panorama Form Selection dialog, displaying all forms. It will store the button selection and form selection in the global variables defined.

```
global buttonname,formname
formselect 2086,0,buttonname,formname
```

This procedure opens a custom Form Selection dialog (# 3000) displaying Print forms only and pre-selects the form called Sheet. The procedure makes a decision based on one of three buttons pressed: Cancel, Print, or Edit.

```
local PrintButton,PrintForm
PrintForm = "Sheet"
openresource "Dialogs"
formselect 3000,2,PrintButton,PrintForm
if PrintButton = "Cancel"
    stop
endif
if PrintButton = "Print"
    openform PrintForm
    print dialog
    closewindow
endif
if PrintButton = "Edit"
    openform PrintForm
    graphicsmode
    stop
endif
```

Reading and Modifying Form Comments in a Procedure

Using the `formcomments()` function a procedure can read the form comments from any procedure in any open database. This function has two parameters, `database` and `form`. Here is an example that checks for the word `printable` in the comments for the current form, and only prints the form if the comments contain this word.

```
if formcomment(info("databasename"),info("formname")) contains "printable"
    print dialog
else
    message "Sorry, this form is not printable."
endif
```

A procedure cannot set the value of the form comments directly, but using the `formcomments` statement it can pause and allow the user to type in form comments.

Accessing and Modifying Procedures

Procedures are usually accessed and edited manually using Panorama's built in procedure editor, but a procedure can also access, modify and create other procedures automatically.

Use the `dbinfo()` function to find out what procedures are in any open database.

```
dbinfo("procedures",database)
```

For the database parameter you can specify any open database or specify "" for the current database. The result of this function is a carriage return delimited list of the procedures in the specified database.

Here is a procedure that opens a database based on a parameter passed to the procedure. It then checks to see if the newly opened database contains a procedure named `.InitializeWidget`. If it does, it calls the procedure, otherwise the `call` statement is skipped.

```
openfile parameter(1)
if arraycontains(dbinfo("procedures","", ".InitializeWidget"),cr())
    call .InitializeWidget
endif
```

Accessing a Procedure's Source Code

The `getproceduretext()` function returns the source code of any procedure in any open database (assuming your security access level allows you to see the procedure). This function has two parameters, the name of the database and the name of the procedure:

```
getproceduretext(database,procedure)
```

If the database name is "" the current database will be used.

This example creates a list of all procedures in the current database that contain a `functionvalue` statement.

```
local plist
plist=dbinfo("procedures","")
arrayfilter plist,plist,cr(),
    ?(getproceduretext("",import()) contains "functionvalue",import(),"")
arraystrip plist,cr()
```

In other words, this is a list of all procedures in this database that can be called with the `call()` function.

Changing a Procedure's Source Code

The `setproceduretext` statement allows a procedure to change the source code of the current procedure. The procedure must already have been opened, either manually or with the `openprocedure` or `goprocedure` statements. The `setproceduretext` statement simply replaces all of the text in the procedure. Any existing text in the procedure will be wiped out.

The `setproceduretext` statement has one parameter, the new source code text.

```
setproceduretext sourcetext
```

Creating a New Procedure

To create a new procedure use the `makenewprocedure` statement. This statement has two parameters:

```
makenewprocedure name,before
```

The `name` parameter is the name of the new procedure. The name can be up to 25 characters long and must be unique (you cannot use the name of a procedure that already exists in this database.)

The optional **before** parameter specifies the location of the new procedure in the procedure list. If this parameter is missing or is "" the procedure will be added at the end of the list. If **before** contains a valid name of an existing procedure the new procedure will be added before the specified procedure.

This example creates a new procedure named **Grand Total**. In the View menu this procedure will be at the end of the list.

```
makewindow "Grand Total"
```

This example also creates a new procedure named **Grand Total**, but in the View menu this procedure will be at the beginning of the list.

```
makewindow "Grand Total",firstline(dbinfo("procedures",""))
```

This example creates a new procedure named **Grand Total** and fills in the source code of the procedure with code to total the **Debit** field.

```
makewindow "Grand Total"
openprocedure "Grand Total"
setproceduretext "field Debit"+cr()+"Total"
closewindow
```

Storing Procedures in a Dictionary

Panorama provides several statements that allow you to copy one or more procedures into a dictionary (see “[Data Dictionaries](#)” on page 602). This allows you to store procedures separately from the database, or to transfer procedures from one database to another. The Panorama Enterprise server uses this technique to transfer databases from the client to the server.

The **saveallprocedures** statement saves all of the procedures in a specified database into a variable as a dictionary.

```
saveallprocedures database,variable
```

Once saved this way you can access individual procedures with the **getdictionaryvalue()** function.

In addition to saving all procedures you can also store only open procedures

```
saveopenprocedures database,variable
```

Or just a single procedure:

```
saveoneprocedure database,procedure,variable
```

Once you have a variable that contains a dictionary of procedures you can load these procedures back into the original database (essentially a revert to saved for procedures) or load them into another database. This can provide a quick way to transfer multiple procedures from one database to another. To load the procedures use the **loadallprocedures** statement:

```
loadallprocedures variable,procedurelist
```

The **variable** parameter is the variable that contains the dictionary. The **procedurelist** parameter is also the name of a variable. This variable will be set to a carriage return delimited list of the procedures that were actually changed by the **loadallprocedurestatement**. If no procedures were changed then the list will be empty. This second parameter is optional — if you don't care what procedures were modified you can omit this parameter.

Here is an example that copies the `.ModifyRecord` procedure from the current database to the `Contacts` database.

```
local pxfr,pupdates
saveoneprocedure "", ".ModifyRecord",pxfr
openfile "Contacts"
loadallprocedures pxfr,pupdates
if pupdates=""
    message ".ModifyRecord was already copied"
endif
```

The `comparedictprocedures` statement allows you to compare procedures that have been saved into a dictionary with an open database. It returns information about which procedures have changed.

```
comparedictprocedures database,dictionary,modified,new,removed
```

The `database` parameter is the name of the database, or "" for the current database.

The `dictionary` parameter is the variable that contains the dictionary (built with the `saveallprocedures` statement).

The `modified` parameter is a field or variable to receive carriage return delimited list of procedures that were actually modified.

The `new` parameter is a field or variable to receive carriage return delimited list of procedures that exist in the dictionary but not in the database.

The `removed` parameter is a field or variable to receive carriage return delimited list of procedures that exist in the database but not in the dictionary.

Our example is in two parts. The first part saves the current procedures into a permanent variable.

```
permanent savedProcedures
saveallprocedures "",savedProcedures
```

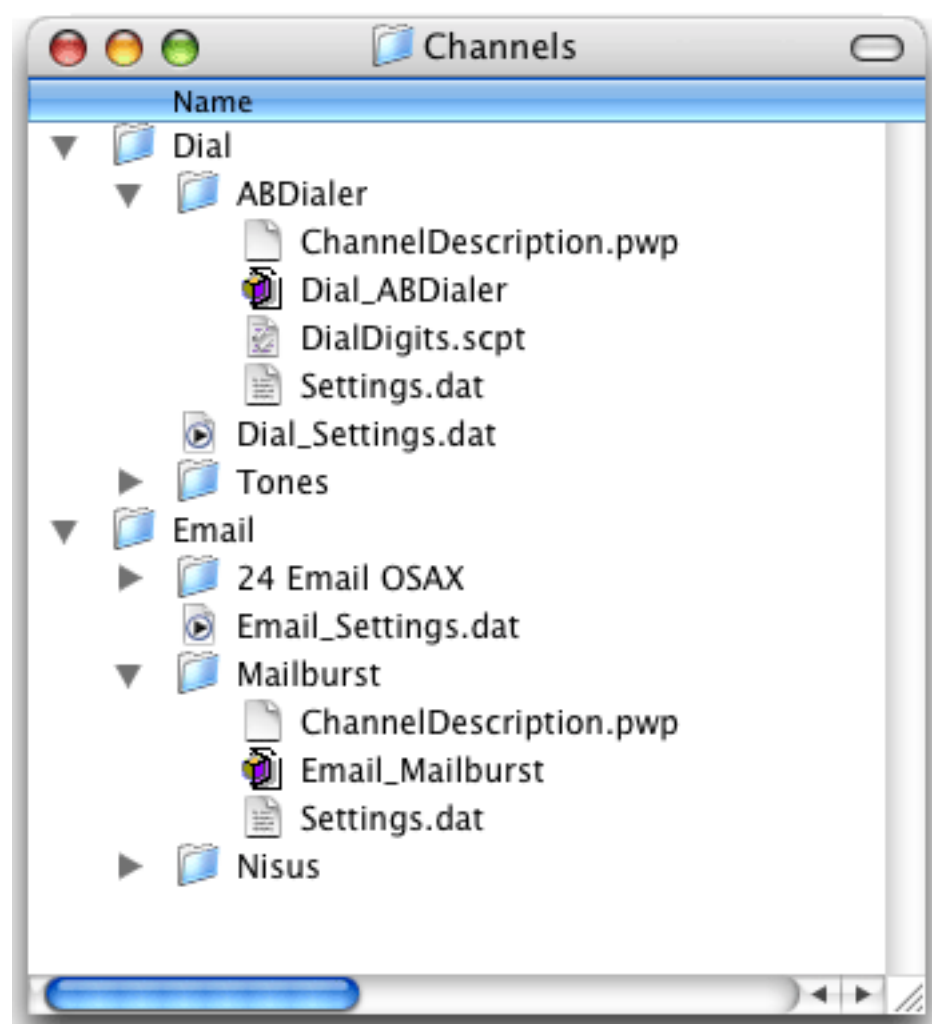
Use this procedure to save the current state of the procedures. Then do some work, editing various procedures, creating new ones, deleting old ones. After doing this you can use the following procedures to see which procedures were changed.

```
local modded,added,xed
comparedictprocedures "",savedProcedures,modded,added,xed
displaydata "Modified Procedures:"+cr()+modded+cr()+cr()+
    "New Procedures:"+cr()+added+cr()+cr()+
    "Deleted Procedures:"+cr()+xed
```

Writing Your Own Channel Modules

Panorama comes with a number of channel modules for sending e-mail, dialing the phone, and interfacing with other web sites and third party software. If you have programming experience you can write your own channel modules. To help make this easier we have created a **Channel Workshop** wizard that will create the core of your new module for you. The next few sections provide an overview of this process, but before you write your own new channel we recommend that you take some time to study the modules provided with Panorama.

Each module is contained in a folder that has the name of the module. This folder must contain a database that has the name of the channel and the name of the module, separated by an underscore (_). This is called the *module database*. The illustration below illustrates two of the modules that come with Panorama: [ABDialer](#) and [Mailburst](#). (If you use the **Channel Workshop**, described below, it will automatically create the correct folders and database for you.)



As you can see, the module folder must be placed inside the folder for the corresponding channel type. In other words, dialing modules must be in the [Dial](#) folder, email modules, in the [Email](#) folder, etc.

In the illustration above, each folder contains a .pwp and .dat file. These files are created automatically by the **Channels** wizard. The [ABDialer](#) folder also contains an AppleScript file used by the module. You should store any additional files needed by the module within the same folder as the module itself.

The ModuleInformation Procedure

Every module database must contain a procedure named [ModuleInformation](#). Panorama calls this procedure to find out about the settings and capabilities of this module. The procedure has two parameters. Parameter 1 is a text parameter that specifies what information Panorama is requesting. The procedure must return this information in the second parameter using the setparameter statement. Unless noted otherwise the information must be returned in data dictionary format (see “[Data Dictionaries](#)” on page 602). To create a dictionary first create a variable and assign it a value of "". Then use the [setdictionaryvalue](#) name,value statement to add each name/value pair to the dictionary.

The types of information that Panorama will request are "settings", "settingnames", "settingdefaults", "settingdescriptions", and "url". The `ModuleInformation` procedure will usually use a case statement to process each of these options.

Type	Description
"settings"	The procedure must return a carriage return separated array listing the settings used by this module. The settings should not contain any spaces, punctuation or special characters.
"settingnames"	The procedure must return a dictionary that contains the settings and setting names. The setting name may contain spaces, punctuation, etc. For example, the AreaCode setting might have a name of Area Code.
"settingdefaults"	The procedure must return a dictionary that contains the settings and default values for each setting.
"settingdescriptions"	The procedure must return a dictionary that contains the settings and default values for each setting.
"url"	The procedure must return a dictionary that contains the URL's to be listed in the URL wizard. In each name/value pair the name is the URL itself and the value is a short description of the URL (perhaps the page title).

You don't have to write the `ModuleInformation` from scratch yourself. Instead, you can use the `Channel Workshop` wizard (described below) to write the procedure for you.

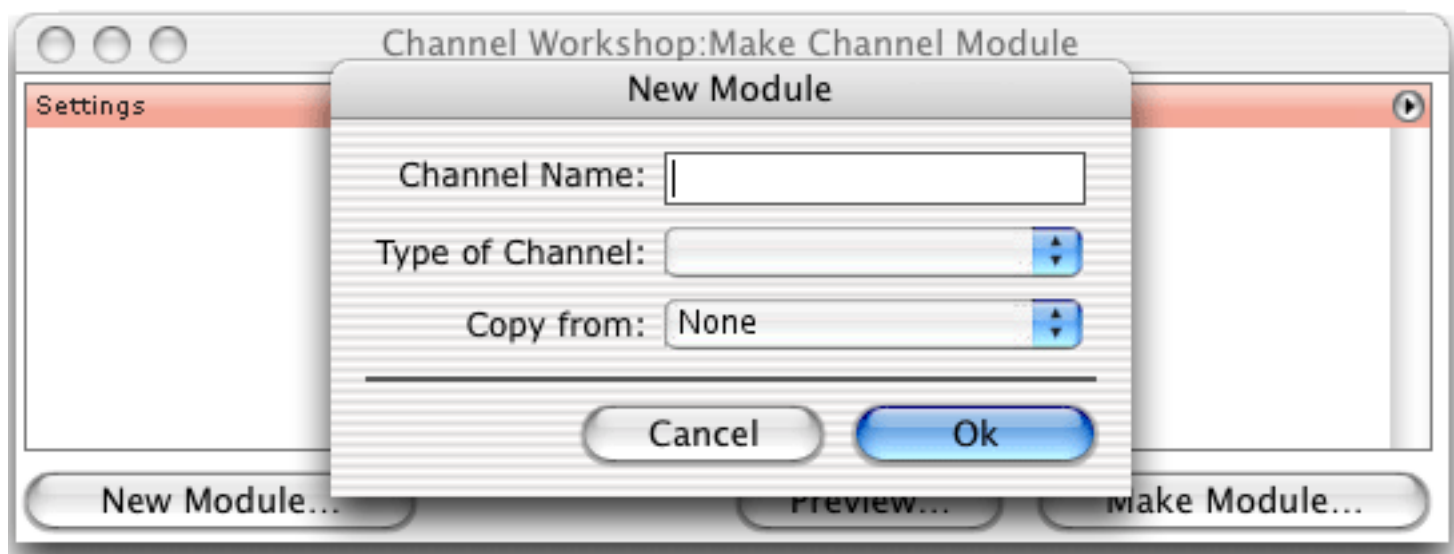
Channel Specific Procedures

Each type of channel will require one or more additional procedures to perform the work of the channel.

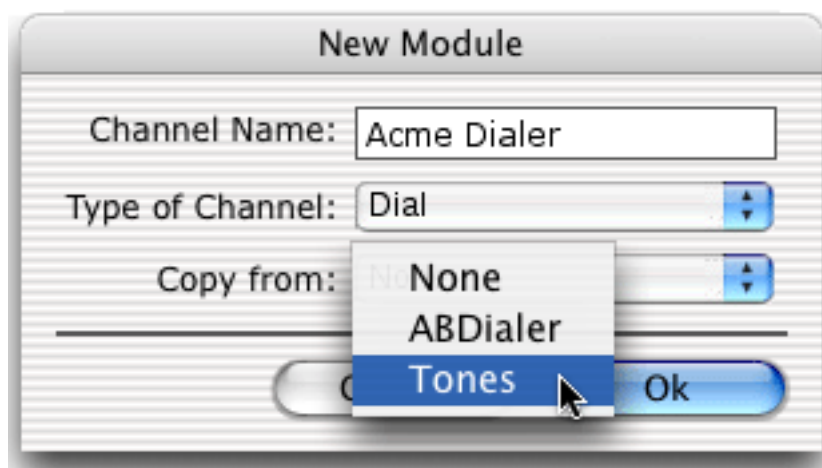
Type of Channel	Procedure Name	Description
Dial	DialDigits	This procedure has one parameter - the digits to dial. The procedure should dial these digits exactly, with no additions or subtractions.
Email	SendEmail	This procedure sends an e-mail. The procedure has two parameters: <code>options</code> and <code>message</code> . The <code>options</code> parameter must contain assignments for various e-mail options, including the recipient, sender, subject, server, etc. For more information see the documentation for the <code>SendMail</code> statement in the Programming Reference Wizard. The <code>message</code> parameter will contain the body of the e-mail message to be sent.

The Channel Workshop Wizard

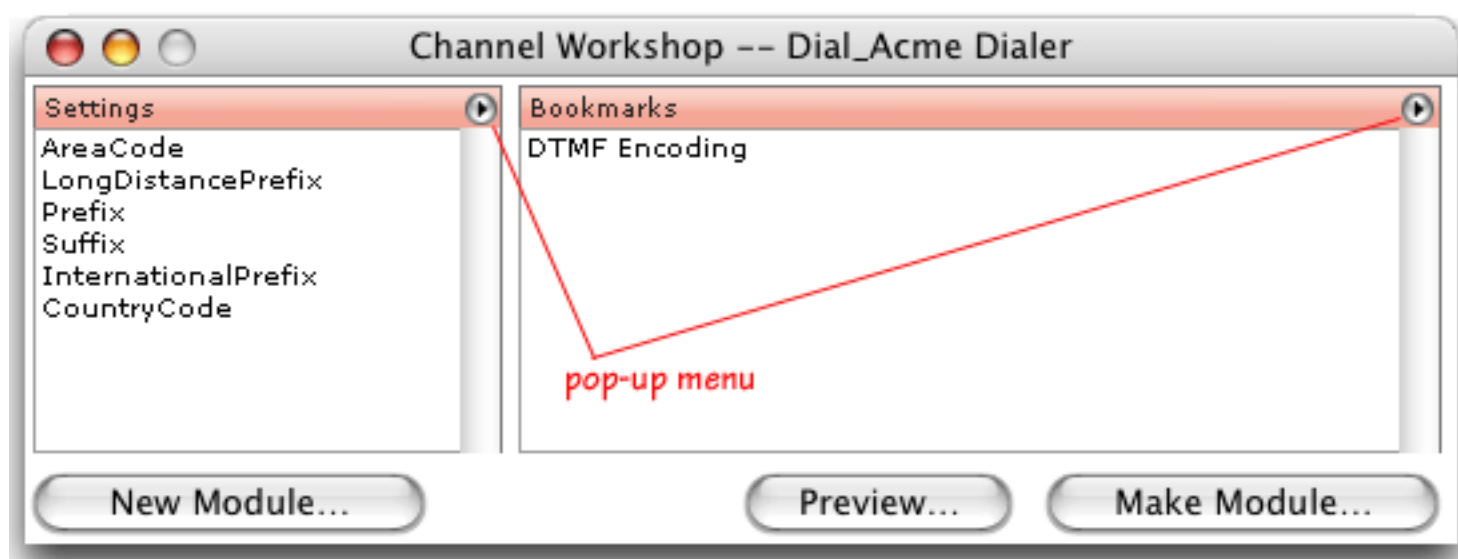
The **Channel Workshop** wizard automates much of the drudgery of creating a channel module. Of course the workshop can't write all of your code for you (it does write some of it) but it does automate the basic tasks required to create a channel, freeing you to focus on writing the actual code necessary to implement the channel. When you first open the **Channel Workshop** it automatically begins the process of creating a new channel.



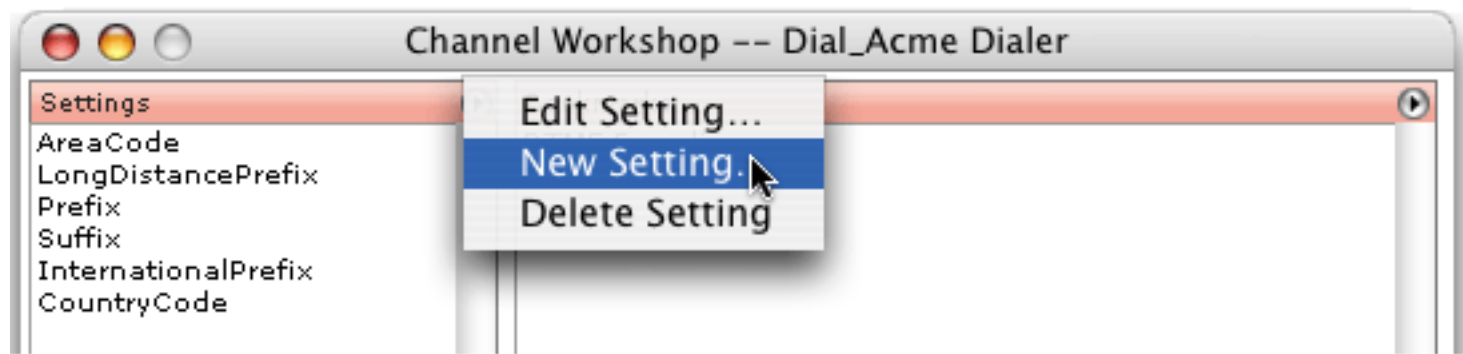
Your first step is to type in a name for the new channel, and to use the pop-up menu select the type of channel. Once you have selected the type of dialer you may want to use the bottom pop-up menu to initialize your new module with settings from an existing channel.



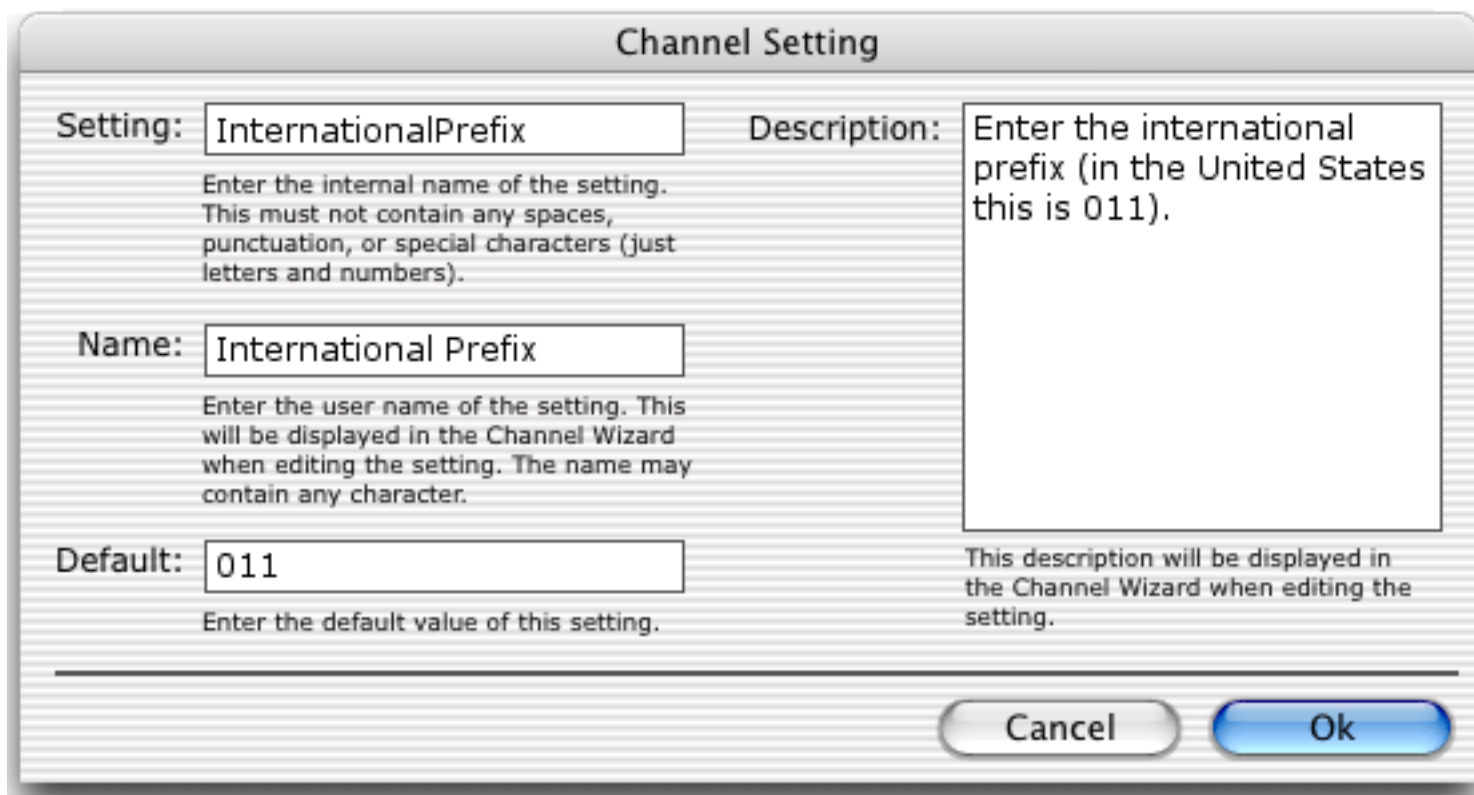
Press **Ok** to start configuring your new module. Since we selected the **Copy from** option we'll start with some settings already set up.



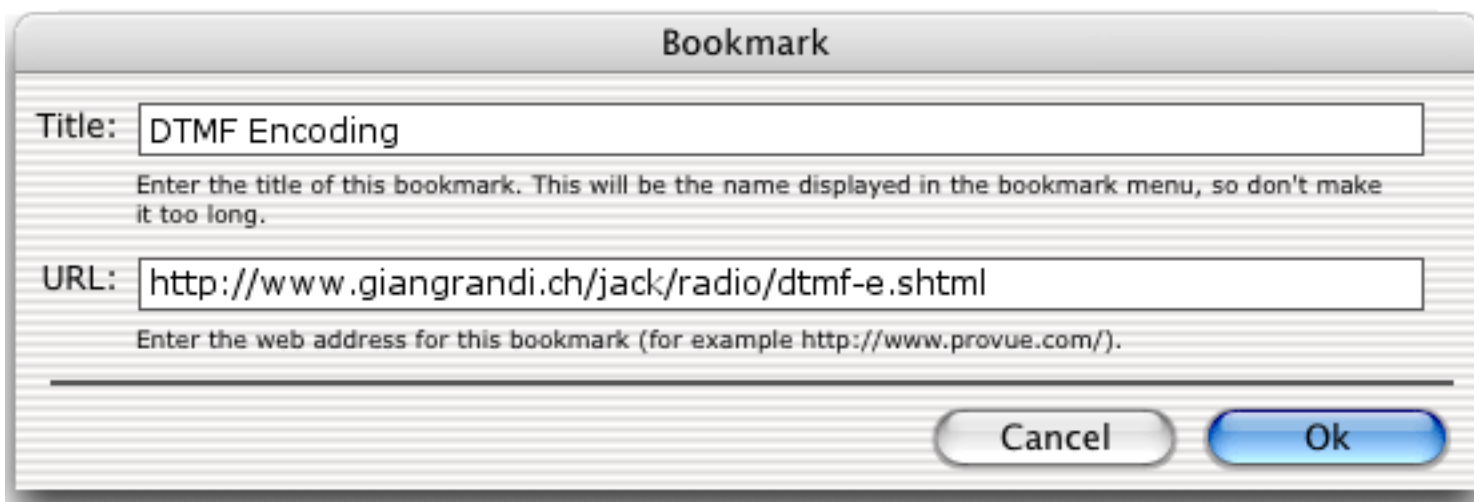
Use the pop-up menu in the upper right hand corner to add, delete, or edit settings.



You can also double click on a setting to edit it.



You can modify the bookmarks the same way.



You can also add bookmarks by dragging from your browser into the bookmark area.

Previewing the ModuleInfo Procedure

Press the **Preview** button if you'd like to see a preview of the code of the **ModuleInfo** procedure you are about to generate.

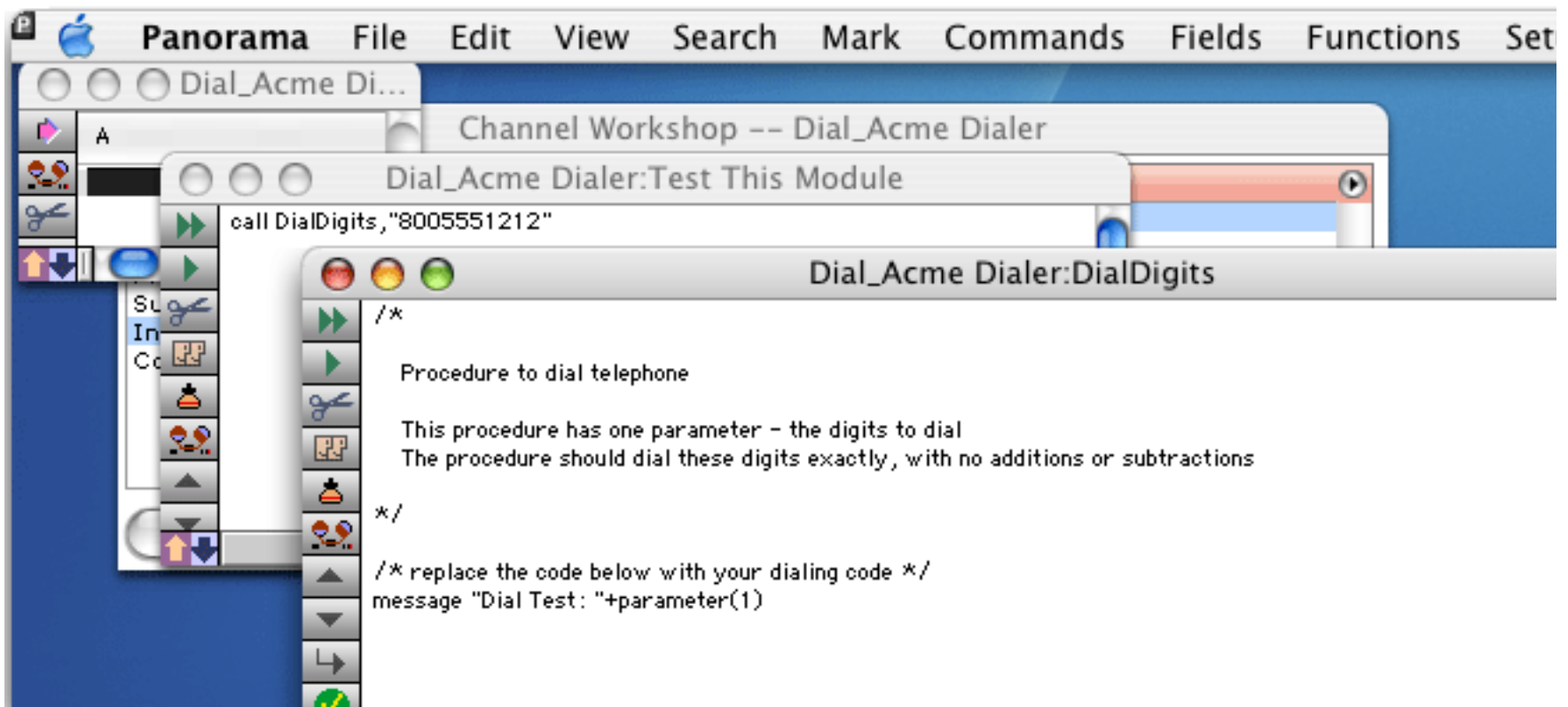


```
case parameter(1) match "settings"
  requestedinfo=" AreaCode "+¶+" LongDistancePrefix "+¶+" Prefix "+¶+" Suffix "+¶+" InternationalPrefix "+¶+" CountryCode "
case parameter(1) match "settingnames"
  initializedictionary requestedinfo,
    " AreaCode ", " Area Code ",
    " LongDistancePrefix ", " Long Distance Prefix ",
    " Prefix ", " Prefix ",
    " Suffix ", " Suffix ",
    " InternationalPrefix ", " International Prefix ",
    " CountryCode ", " Country Code "
case parameter(1) match "settingdefaults"
  initializedictionary requestedinfo,
    " AreaCode ", " ",
    " LongDistancePrefix ", " 1 ",
    " Prefix ", " ",
    " Suffix ", " ",
    " InternationalPrefix ", " 011 ",
    " CountryCode ", " "
case parameter(1) match "settingdescriptions"
  initializedictionary requestedinfo,
    " AreaCode ", "Enter your local area code (for example 213).",
    " LongDistancePrefix ", "Enter the long distance prefix (in the United States and Canada this is 1).",
    " Prefix ", "Enter digits to be dialed before every phone number (for example for a PBX or calling card).",
    " Suffix ", "Enter digits to be dialed after every phone number (for example for a calling card).",
    " InternationalPrefix ", "Enter the international prefix (in the United States this is 011).",
    " CountryCode ", "Enter your country code (in the United States this is 1).",
case parameter(1) match "url"
  initializedictionary requestedinfo,
    "http://www.giangrandi.ch/jack/radio/dtmf-e.shtml", "DTMF Encoding "
endcase
```

It's not actually necessary to ever look at this code, but it does give you an overview of the configuration you are setting up.

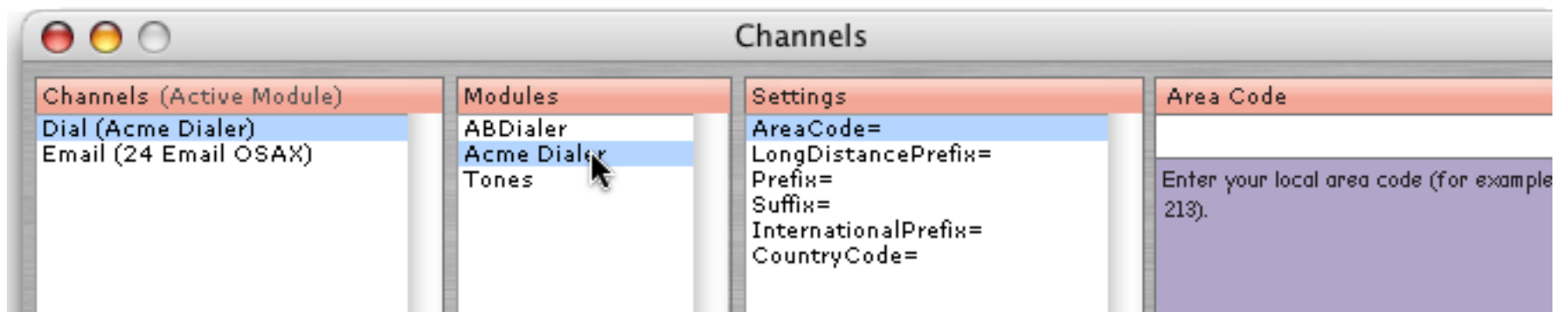
Creating the Module

When you are ready to create the module press the **Make Module** button. As shown below, Panorama will create the new module in the correct location (making any new folders that are necessary). It will then open the new module for you so you can immediately begin working on the code for your new module. To test your new module, bring the **Test This Module** window to the front and press **Command-R**.



Tip: If your procedure needs to use an AppleScript to control another application don't forget about the new **ExecuteAppleScript** statement, which allows you to build and execute an AppleScript on the fly.

Once your new module is finished you can select and configure it with the **Channels** wizard, just like any other channel module.



Chapter 4: Working With Alternate Programming Languages



Panorama isn't limited to Panorama's own built-in programming language (described in Chapters 24 and 25 of the Panorama Handbook). When using Panorama on OS X a programmer can also embed code written in several other languages into a Panorama procedure, including AppleScript, Unix Shell Scripts, Perl, Ruby, Python and PHP.

Embedding alternate programming language code allows Panorama to tackle jobs it couldn't handle on its own. It also allows you to avoid re-inventing the wheel by leveraging the code libraries available for these alternate languages. Some examples of the libraries available for Perl, Ruby, Python and PHP include tools for manipulating complex numbers and matrices, pattern matching, statistical calculations, linear equations, image manipulation, sql database access, cryptographic services, xml parsing and various internet protocols including dns, ftp, smtp, pop3, imap, telnet, ssh, bittorrent, soap and low level tcp/udp access. Some of these tasks *could* be done with Panorama, but using a pre-built library is a lot easier! We'll have a few examples that illustrate using a pre-build library later in this chapter.

Choosing a Language

With multiple alternate languages to choose from, how do you pick the one you want to use? In some cases the choice is clear, in other cases it may be more of a personal preference. We'll start this chapter with a very basic summary of the each language available for embedding within Panorama. If you are already familiar with these languages and know which one you are going to use you may want to skip ahead to "[Code Embedding 101](#)" on page 754.

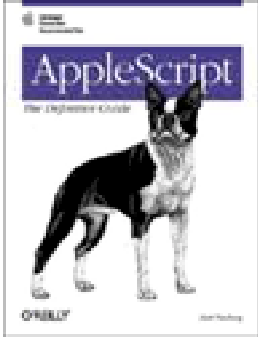
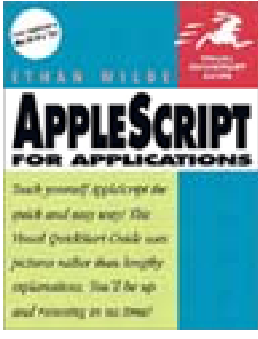
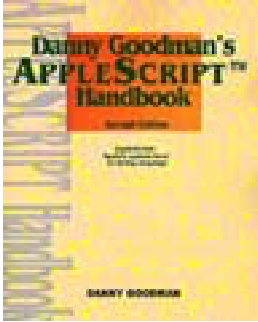
Keep in mind that entire books have been written each of these programming languages, in some cases dozens of books. Teaching how to program in each of these languages is beyond the scope of this book. Depending on the task you are tackling you will need to pick up some knowledge about the languages you choose to use. We'll provide some recommendations about web sites and books that we've found useful, you can find more at your local bookstore, Amazon, or with a search engine (Google, etc.).

AppleScript

AppleScript is a special language invented by Apple to allow scriptable applications to work together. If an application is scriptable then this language can be used to control it and communicate with it. Not all Macintosh applications are scriptable, but there are dozens of significant applications that are, including Address Book, iCal, Finder, Mail.app, iTunes, Safari, Microsoft Word and Excel, Adobe Photoshop, Illustrator and InDesign and many more. For example Panorama can use an embedded AppleScript to retrieve a table from Excel (or copy a table *to* Excel), or to look up a name from your Address Book.

AppleScript uses an English-like syntax that is supposed to be simple and natural but unfortunately can trip you up. Learning AppleScript is further complicated by the fact that each scriptable language has its own extensions to the language that bring their own quirks and traps. That said, AppleScript is a powerful tool that lets you combine the power of two or more programs together even if they were never intended to work together by their original authors. In appropriate situations AppleScript can make daunting tasks routine and is one of the features that make the Macintosh a uniquely powerful system (unlike the other embeddable languages discussed in this chapter, AppleScript is unique to Macintosh systems and is not available for any other type of computer).

Listed below are several books that we've found especially helpful for learning and mastering AppleScript:

	<p>AppleScript : The Definitive Guide</p> <ul style="list-style-type: none"> • Author: Matt Neuburg • Paperback: 590 pages • Publisher: O'Reilly; 2nd edition (January 4, 2006) • ISBN: 0596102119 <p>If you've already got some programming experience, this book is the ultimate reference for AppleScript. Rather than rely on published documentation, the author did his own testing to uncover all the nooks and corners of this complex language, and he doesn't hesitate to point out the shortcomings as well as the strengths of the AppleScript system. Be sure to pick up at least one other AppleScript book, because this one covers only the language itself, and does not include any specifics of how to use AppleScript with other applications.</p>
	<p>AppleScript For Applications</p> <ul style="list-style-type: none"> • Author: Ethan Wilde • Paperback: 480 pages • Publisher: Pearson Education; 1st edition (November 15, 2001) • ISBN: 0201716135 <p>This book takes a cookbook approach to AppleScript. The book includes complete ready-to-use scripts for popular Mac apps such as Word, Photoshop, and QuarkXPress plus a companion Web site for downloads and helpful links. These scripts can be used as is or as a starting point for your own custom scripts. It also includes a step-by-step beginner's tutorial to getting started with AppleScript.</p>
	<p>Danny Goodman's AppleScript Handbook</p> <ul style="list-style-type: none"> • Author: Danny Goodman • Paperback: 576 pages • Publisher: iUniverse; 2nd edition (April 1, 2000) • ISBN: 0966551419 <p>This is the classic AppleScript book, originally published many years ago when AppleScript was first released. It's a bit dated, but still a good resource.</p>

If you are serious about AppleScript we highly recommend that you purchase a copy of **Script Debugger** by Late Night Software. As mentioned above, AppleScript can be quite perverse. Sometimes working with AppleScript can feel like trying to do carpentry in a shop with no windows and no lights — you can only work by feel! When we started working with Script Debugger it was as if the light switch had been flipped on. Script Debugger is not cheap but if your time is at all valuable the payback period is incredibly short — and the quality of your code will improve as well. Script Debugger is discussed in more detail later in this chapter, see also.

Shell Scripts

Shell scripts are the lingua franca of all UNIX and Linux systems, with Mac OS X being no exception. Shell commands were the original method for operating UNIX systems. Instead of looking at windows or clicking the mouse the user would type commands onto a command line. Each command would perform a specific task, for example changing to a different folder, displaying a list of files, modifying a file (or files), connecting to a network, etc. In order to use the system you needed to memorize the most common shell commands and their options, while having a manual or reference book handy for the less often used commands. Unless you were a computer geek you probably wouldn't think this was a very user friendly system, and in fact computers really didn't take off until graphical mouse based control systems were added.

Shell commands, however, do have some advantages over graphical mouse based systems. While shell commands are harder to use, they are much easier for the programmer to create. Because of this there are many operations that can be performed with shell commands that have never been enhanced with a graphical control. These operations can only be performed with a shell command.

Another big advantage of shell commands is that they were designed to be easy to automate. In fact, anything that can be done with a shell command can be automated. That's not true for mouse based controls, which often can't be automated at all.

On Mac OS X shell commands can be manually entered using Apple's **Terminal** program. When you press the **Return** key any output is displayed below the command.

The screenshot shows a Terminal window titled `/usr/bin/login (tty2)`. The prompt is `[localhost:~] chris%`. The user enters the command `cd Misc\ Docs/`. The prompt changes to `[localhost:~/Misc Docs] chris%`. The user enters the command `ls -l | grep "Aug 1[0-9]"`. The output is:

```
-rw-r--r-- 1 chris  staff   503808 Aug 10 09:27 8-10 meeting notes.doc
-rw-r--r-- 1 chris  staff   503808 Aug 15 02:27 FilesForCraig.zip
-rw-r--r-- 1 chris  staff  1105920 Aug 18 08:13 ForJim.sit
-rw-r--r-- 1 chris  staff   991232 Aug 17 08:21 Layout Comp.pdf
-rw-r--r-- 1 chris  staff   544768 Aug 18 21:01 Output report.txt
```

The command line and the output are circled in red. A red arrow points from the text "shell command" to the command line, and another red arrow points from the text "output" to the output lines.

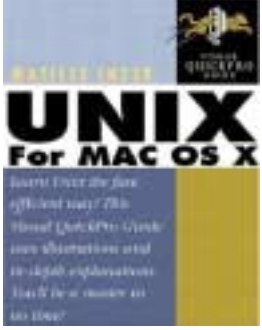
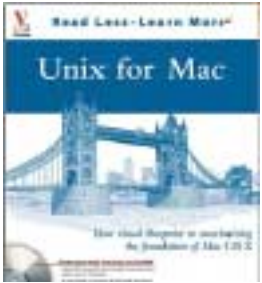
To use shell scripts within Panorama you actually embed the shell command directly into the Panorama procedure. Instead of being displayed, the output is routed to a Panorama field or variable.

If you're new to shell scripting you'll probably find that the biggest challenge is knowing what command you need to use to perform the task you need done. There are literally thousands of commands available, and most have cryptic names like **cd** (change directory), **ls** (list files), **cp** (copy files) etc. There is a complete list of commands available on Mac OS X at this web page:

<http://developer.apple.com/documentation/Darwin/Reference/ManPages/>

Each command listed on this page is a link — click the link for detailed information about that particular command.

There are many books available on the topic of shell scripting, both for Mac OS X and for general UNIX/Linux systems. Quite frankly we haven't found any one book that is really comprehensive. Here at ProVUE we've wound up with more than a handful of various shell scripting books and picked up tidbits from each. That said, here are a couple that you may want to consider.

	<h3>UNIX for Mac OS X: Visual QuickPro Guide</h3>
	<h3>Unix for Mac</h3>

- Author: Matisse Enzer
- Paperback: 560 pages
- Publisher: Peachpit Press; 1st edition (December 3, 2002)
- ISBN: 0201795353

This book shows how to configure the Unix environment, navigate permissions, directories and files, run different Unix utilities, configure and run the Apache Web server, and much more. It also covers how to protect files with Unix's security tools, and how to fix problems when things go wrong..

- Author: Sandra Henry Stocker , Kynn Bartlett
- Paperback: 352 pages
- Publisher: Visual (March 7, 2003)
- ISBN: 076453730X

Includes step-by-step screen shots demonstrating more than 160 Unix tasks, including: using the terminal application; navigating the file system; customizing the shell and writing shell scripts. It avoids bogging the reader down with excessive information while, at the same time, providing a good introduction.

There are also many web sites with shell scripting tutorials. Some of the ones we've found start out well but don't go very far. If you find a good one let us know and we'll pass it along in future versions of this handbook!

Scripting Languages

The four remaining languages (Perl, Ruby, Python and PHP) are commonly called “*scripting languages*.” Though of course each of these languages is unique, they also have a number of features in common.


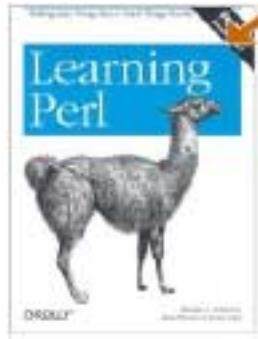

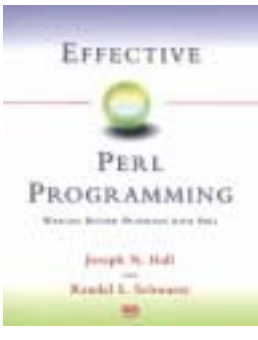
Feature	Notes
Interpreted Execution	Each of these languages is interpreted rather than compiled. That means that there is no preparation necessary to run your programs, just edit and go (unlike compiled languages like C or Java).
Dynamic Typing	Unlike strongly typed languages like C and Java, these languages don't require that you define the data types of variables in advance. In fact you can even change the data type of a variable on the fly as the program runs.
Automatic Memory Management	Unlike lower level languages like C, these languages handle all memory allocation and management for you.
Text Handling	Each of these languages has powerful tools for working with text and doing pattern matching.
Dynamic Arrays (Lists)	All of these languages allow arrays to be extended dynamically as the program runs instead of requiring that arrays be defined statically during compilation. They also support <i>associative arrays</i> (hashes) that allow array values to be indexed by name instead of numerically.
Objects	All of these languages allow for the creation of classes and objects for modern modular “object oriented” programming.
Extensive Libraries	Each of these languages has vast (and growing) libraries of ready to use modules for performing useful tasks. Before programming something yourself check the libraries — you may find that some or all of the work has already been done for you.

Each of these languages has its own group of adherents (to put it politely in some cases). Perl and PHP have been popular for many years, but Ruby and Python have been coming on strong recently (and all of these languages are constantly evolving, transitioning to new versions and generally jockeying for position). To a large degree your choice depends on personal preferences and upon the library packages you need and code that may already be available to you. The quick summaries below are just our impressions, on the internet you'll find as many opinions as you find commentators. (By the way, you don't have to pick just one of these languages, if you needed to you could embed all six of these languages into a single Panorama procedure!)

Perl

Perl is the granddaddy of scripting languages — the first version appeared in 1987. Perl was originally developed for text and report processing (the name stands for Practical Extraction and Report Language), but today some people refer to it as “the duct tape of the internet” because while it may not always be the neatest solution it can (and has been) used for almost anything. Perl has so many features it is kind of like the kitchen sink of programming — in fact the Perl motto is “there’s more than one way to do it!” (Panorama’s programming language could have the same motto, which makes us feel quite comfortable with Perl.) Perl wasn’t originally object oriented, but current versions have supported object oriented programming for quite some time.

There are many books and online resources available for Perl. Here are a few of our favorites:

	<p>Programming Perl</p> <ul style="list-style-type: none"> • Author: Larry Wall, Tom Christiansen, Jon Orwant • Paperback: 1092 pages • Publisher: O'Reilly Media, Inc.; 3 edition (July 14, 2000) • ISBN: 0596000278 <p>We probably shouldn't list this book first, but we couldn't help ourselves. Programming Perl is clear, comprehensive and witty. It's now the standard we use to measure our own writing (unfortunately coming up well short), and it is currently our favorite programming book. First published in 1990, it is considered the “bible” of Perl. What's not to like? At over 1,000 pages it may be a bit <i>too</i> comprehensive for many. But if you really want' to learn Perl inside and out, this is <i>the</i> book to go to.</p>
	<p>Learning Perl</p> <ul style="list-style-type: none"> • Author: Randal L. Schwartz, Tom Phoenix, Brian D Foy • Paperback: 304 pages • Publisher: O'Reilly Media, Inc.; 4th edition (July 14, 2005) • ISBN: 0596101058 <p>If Programming Perl is too much for you then you might want to try this one.</p>
	<p>Perl Cookbook</p> <ul style="list-style-type: none"> • Author: Tom Christiansen, Nathan Torkington • Paperback: 927 pages • Publisher: O'Reilly Media, Inc.; 2nd edition (August 2003) • ISBN: 0596003137 <p>When you just need to solve a problem right now this book is the one to turn to. It's filled with ready to use code for problems ranging from the simple (reformatting paragraphs) to complex (sending e-mail with attachments). Each cookbook “recipe” is explained and analyzed, and all of the code is available online so that you can simply paste it right into Panorama.</p>
	<p>Effective Perl Programming: Writing Better Programs with Perl</p> <ul style="list-style-type: none"> • Author: Joseph N. Hall, Randal Schwartz • Paperback: 288 pages • Publisher: Addison-Wesley Professional; 4th edition (December 30, 1997) • ISBN: 0201419750 <p>This book distills years of Perl experience into a book that is both fluid and fun to read. It's somewhat like reading the Perl FAQ; even when you think you know everything, there's more you don't know. Packed with examples and code snippets.</p>

There are numerous Perl resources available on the web. Probably the best places to start are the official Perl web site and the Perl documentation site:


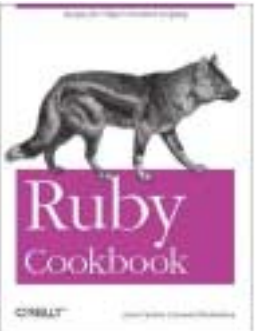

<http://www.perl.org/>

<http://perldoc.perl.org/>

Ruby

Ruby was designed to be a pure object-oriented language from the ground up, so everything—every string, every number, every user-defined data type—all of these are objects in Ruby. Many programmers find Ruby code to be very clean and readable — well known commentator Tim Bray says “Ruby is remarkably, perhaps irresistibly, attractive.”

Some books we have used for studying Ruby are listed below. When looking at Ruby books to help you write Ruby code for Panorama you’ll probably want to avoid books about Ruby on Rails, which doesn’t apply to Panorama.

	<p>Programming Ruby: The Pragmatic Programmers' Guide</p> <ul style="list-style-type: none"> • Author: Dave Thomas, Chad Fowler, Andy Hunt • Paperback: 828 pages • Publisher: Pragmatic Bookshelf; 2nd edition (October 1, 2004) • ISBN: 0974514055 <p>This book (nicknamed the “pickaxe” from the illustration on the cover) is considered by most to be the Ruby “bible.” It includes tutorial-style introduction and overview of the language, a very complete reference and finally, there's lots of information on packages (libraries) you might want to use.</p>
	<p>Ruby Cookbook</p> <ul style="list-style-type: none"> • Author: Lucas Carlson, Leonard Richardson • Paperback: 906 pages • Publisher: O'Reilly Media, Inc. (July 19, 2006) • ISBN: 0596523696 <p>Like the Perl Cookbook described above, this is the book to turn to when you just need to solve a problem right now. It’s filled with ready to use code and each “recipe” is explained and analyzed. All of the code is available online so that you can simply paste it right into Panorama.</p>
	<p>Ruby by Example: Concepts and Code</p> <ul style="list-style-type: none"> • Author: Kevin Baird • Paperback: 326 pages • Publisher: No Starch Press (June 8, 2007) • ISBN: 1593271484 <p>This book uses a real world examples approach to learning Ruby..</p>

Online you can start with the official Ruby web site and site devoted to ruby documentation:

<http://www.ruby-lang.org/en/>

<http://www.ruby-doc.org/>

Python

Python is another language designed from the ground up to be object oriented. It's possibly most famous for the use of indentation to denote programming blocks (instead of begin/end or {/} like most other languages). In other words, whitespace can be significant in a Python program. It's also known for high performance and for extensive, reliable libraries.

Here are the Python books on our bookshelf at ProVUE:

	<p>Learning Python</p> <ul style="list-style-type: none"> • Author: Mark Lutz , David Ascher • Paperback: 552 pages • Publisher: O'Reilly Media, Inc.; 2nd edition (December 2003) • ISBN: 0596002815 <p>In addition to teaching you about Python this book will also get you drinking the Kool-aid — the authors are avid Python evangelists. The first part covers the essentials, including types, operators, statements, classes, functions, modules and exceptions. Once they get through all that there is not much space left for discussion of Python's extensive libraries, though they are briefly covered.</p>
	<p>Programming Python</p> <ul style="list-style-type: none"> • Author: Mark Lutz • Paperback: 1596 pages • Publisher: O'Reilly Media, Inc.; 3rd edition (August 23, 2006) • ISBN: 0596009259 <p>The text covers every conceivable facet of Python, exhaustively exploring the whole language, including Python's extensive libraries.</p>
	<p>Python Essential Reference</p> <ul style="list-style-type: none"> • Author: David M. Beazley • Paperback: 625 pages • Publisher: Sams; 3 edition (February 20, 2006) • ISBN: 0672328623 <p>This reference concisely covers the Python language and libraries. It is designed for readers that are already experienced programmers familiar with other languages like C or Java. If this is you then this book may be a good way to get up to speed with Python without wading through lots of basic material intended for beginners.</p>
	<p>Python Cookbook</p> <ul style="list-style-type: none"> • Author: Alex Martelli, Anna Ravenscroft, David Ascher • Paperback: 844 pages • Publisher: O'Reilly Media, Inc.; 2nd edition (March 18, 2005) • ISBN: 0596007973 <p>Like the Perl and Ruby cookbooks described earlier this book may contain exactly the code you need to solve a problem right now. It's a bit different from the other cookbooks in that the "recipes" have been submitted by a variety of authors (not just the three listed above). Essentially it is an attempt to create an "open source" book. Because of this it is perhaps not quite as organized as the other cookbooks, but there is still a ton of good material here.</p>


The official web site for Python is:

<http://www.python.org/>

PHP

Like Perl, PHP was not originally an object oriented language but object oriented features have been added in recent releases. PHP is especially popular for use on web sites — it is commonly integrated with the popular Apache web server but can also be used separately. Like the other scripting languages here the PHP community has developed extensive code libraries with interesting capabilities like XML processing, SQL database access and image and PDF processing.

So far there is only one PHP book in our library:

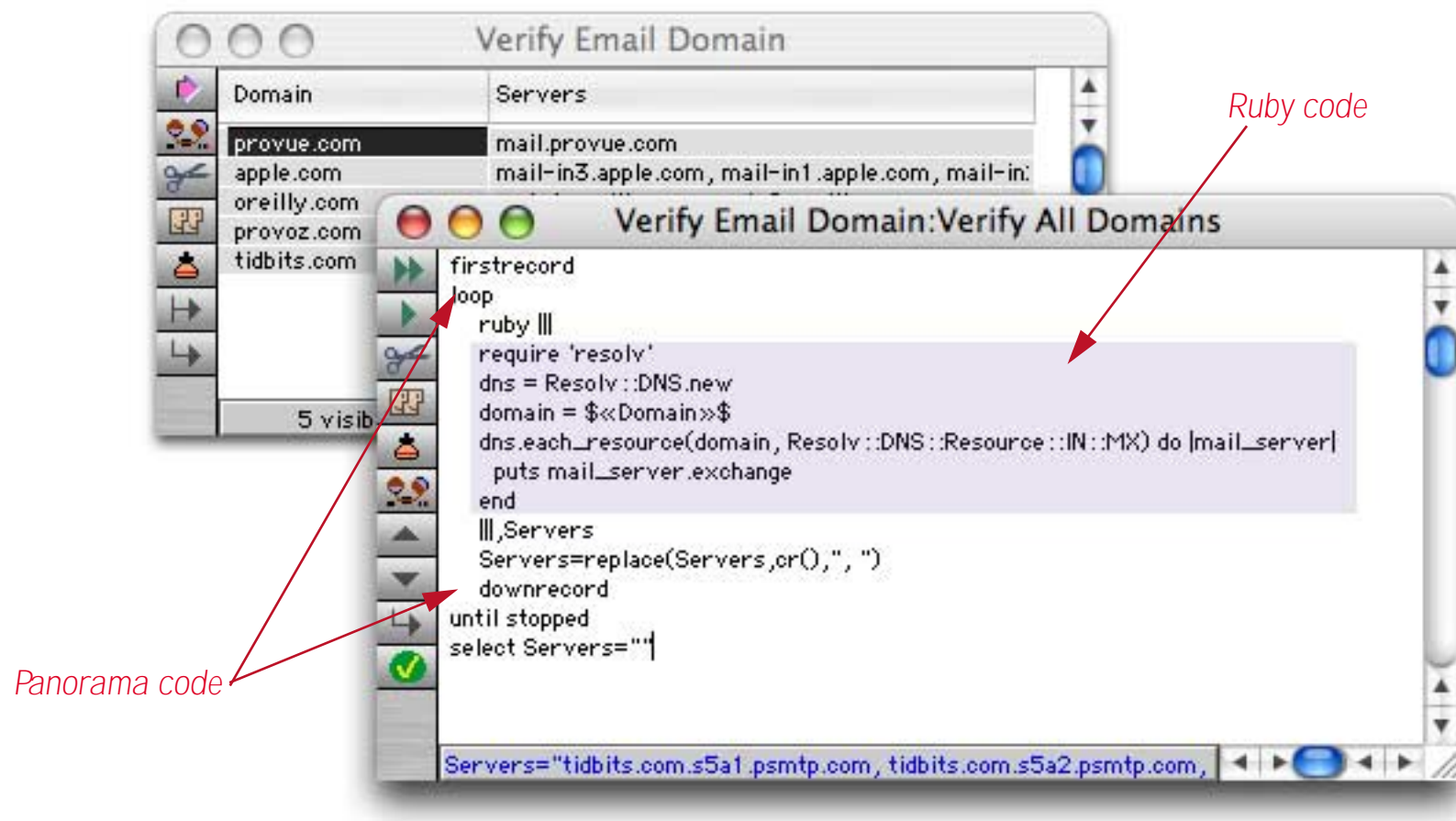
	<h3>Programming PHP</h3> <ul style="list-style-type: none">• Author: Rasmus Lerdorf, Kevin Tatroe, Peter MacIntyre• Paperback: 521 pages• Publisher: O'Reilly Media, Inc.; 2nd edition (April 28, 2006)• ISBN: 0596006810 <p>All of the essential topics (syntax, functions, data types, objects) are covered in the first section of the book. Later sections cover many of the library capabilities available for PHP. Unlike some other PHP books this book is not totally HTML centric, which is good if you're planning to use PHP with Panorama. (You'll still need to adjust many of the examples for use with Panorama instead of within a web page.)</p>
---	--

The official PHP web site is:

<http://www.php.net/>

Code Embedding 101

When we talk about code embedding we mean it literally — the source code in Perl, Ruby, etc. is usually literally embedded in the Panorama source code, as shown in the example below. (In this illustration the code happens to be in Ruby and is highlighted in light purple, but the highlight doesn't appear when you are actually using Panorama.)



Quoting Embedded Code

Embedded code can be in a Panorama variable or even calculated by a Panorama formula, but usually it is simply a text constant (“[Constants](#)” on page 49) as shown in the example above. Since the embedded code itself almost always contains Panorama’s normal constant delimiters like `"`, `'`, `{` and `}` Panorama includes a special delimiter for quoting code - the pipes delimiter. Pipe delimited code starts with a series of pipe (`|`) characters, and terminates with an equal number of pipes. The exact number of pipes to use as a delimiter is up to you on a case by case basis, you can use 2, 3, 4, or more. Within the pipe delimited text you can use any other character you like, including pipes themselves. Here are some examples of text constants using pipes.

```
|||This text is delimited with three pipes||| | |
||||Four pipes here||||
|||Text can even contain |pipes| within the text!|||
```

You can use pipe delimited text anywhere a text constant is called for, but you’ll find them especially useful for quoting code. Here are some additional examples:

```
message perl( |||print "Hello World"||| )
message ruby( |||puts "Hello World"||| )
message python( |||print "Hello World"||| )
```

Embedding Code from a Text File

To embed code from a text file (created with an external editor) use Panorama's `fileload()` function, like this:

```
python(fileload("","mycode.py"))
ruby(fileload("","mycode.rb"))
```

This examples will run code from text files in the same folder as the current database. To learn more about this function see also.

Using Panorama Fields and Variables as Terms in an Embedded Program

Most embedded programs need to get data from Panorama. For example, suppose you embed a Perl program to send an e-mail the Perl program will need to get the e-mail address, subject and body of the message from Panorama. To make this easy to do Panorama allows Panorama fields and variables to be inserted as terms into the embedded code.

To identify that the field or variable is from Panorama rather than a variable belonging to the embedded language you must surround the field or variable name with special tags: `$«` and `»$`. Here's a simple example that copies the contents of a Panorama field named `Email` into a variable in the embedded language named `url`:

```
url=$«Email»$
```

An important point is that you cannot reverse this assignment — a Panorama field or variable cannot be on the left hand side of an assignment statement (in other words, Panorama fields and variables cannot be lvalues). So this assignment statement will not work.

```
$«Email»$=url <!-- will not work!
```

You can, however, use Panorama fields or variables as operands anywhere on the right hand side of an assignment. The Panorama values may be text or numeric values, and you can use multiple Panorama fields or variables in a single expression.

```
Energy=$«Mass»$*c*c
wordlist=$«Abstract»$.split(" ")
label=$«Name»$."\n".$«Address»$."\n".$«City»$.", ".$«State»$." ".$«Zip»$
```

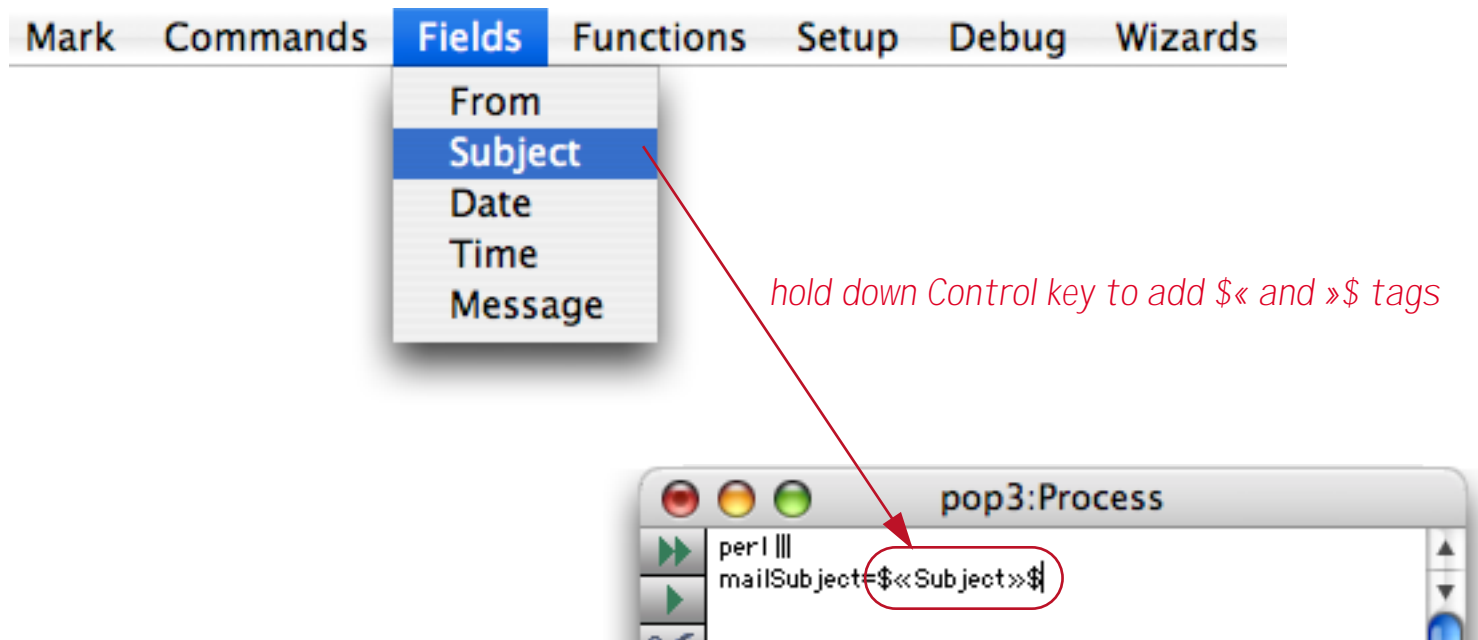
If the Panorama field or variable name includes spaces, punctuation or other special characters you must include a separate set of `«` and `»` characters to the name, like this:

```
Range=$««Tank Capacity»»$*$««Miles Per Gallon»»$
```

The extra set of `«` and `»` characters is necessary to make the text inside the `$«` and `»$` tags a valid Panorama formula. In fact, you can include any Panorama formula inside the tags, as described in a following section.

Using the Field Menu to Insert Fields Names

The **Field** menu inserts a field name into the procedure currently being edited (at the insertion point). If you hold down the **Control** key while using this menu (or use the right button on a two button mouse) the field will be inserted with the special `$«` and `»$` tags necessary for use within embedded programs.



If the Panorama field name includes spaces, punctuation or other special characters the menu will insert a the extra set of `«` and `»` characters necessary (see previous section).

Using a Panorama Formula as a Term in an Embedded Program

The previous sections showed how a Panorama field or variable can be inserted into an embedded program (Perl, Ruby, etc.). This feature isn't limited to just fields or variables, you can actually insert *any* Panorama formula between the `$«` and `»$` tags.

```
fullname = $«upper(FirstName+" "+LastName)»$
fieldlist = $«dbinfo("fields","")»$.split("\n")
$databsepath = $«unixshellpath(dbpath())»$
```

Panorama first calculates the value of the entire formula and then inserts that value as a term in the embedded program.

Getting Data (Results) Back from Embedded Code

Now that you've learned how to get data from Panorama to your embedded code, how do you get data results back from the embedded code to Panorama? There are three possible methods —1) use an embedded code function, 2) using the `ScriptResult` variable, or 3) specifying a field or variable for result. Before discussing these options, however, let's take a moment to talk about how embedded programs generate data results.

Standard Output (stdout)

To output data to Panorama the embedded program simply needs to send data to the standard output stream (often called **stdout**). When using Unix shell commands this happens automatically. In Perl, Python and PHP data is output to stdout with `print`, in Ruby it is done with `puts`. See the documentation for each of these languages to learn more.

The one exception is AppleScript, which doesn't support stdout. When embedding AppleScript the data result returned to Panorama is the value of the final expression executed within the script.

Code Embedding Functions

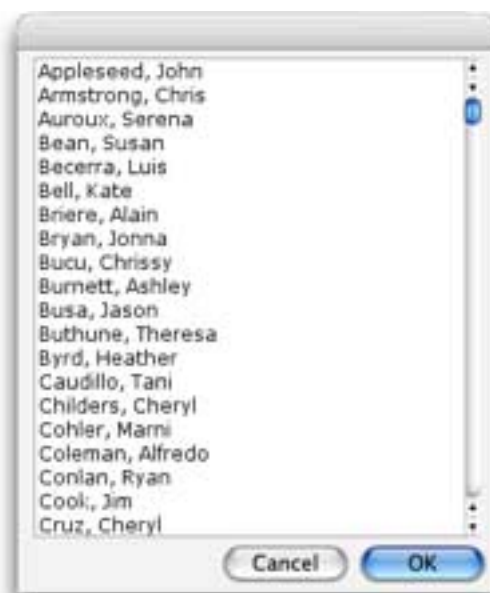
Panorama has six functions that allow alternate programming languages to be embedded within a Panorama formula (which may be used in a procedure or on a form). These are called **code embedding** functions:

```
applescript(code)
shellscript(code)
perl(code)
ruby(code)
python(code)
php(code)
```

Each of these functions takes code written in the specified programming language, executes, and then returns the result. For example this program uses the `applescript()` function to embed a short one line program written in AppleScript (in this illustration the AppleScript code is highlighted in purple, but that does not happen in Panorama's editor).

```
local abNames,Names,namePick
abNames=applescript(|||tell app "Address Book" to get name of every person|||)
arrayfilter tagarray(abNames,{"},"},cr()),Names,cr(),
    lastword(import())+", "+firstword(import())
arraysort Names,Names,cr()
superchoicedialog Names,namePick,{height=360 width=120}
```

When the procedure runs it will run the embedded AppleScript to get the name of every person contained in your Address Book. It will then format the and sort the list, and display it as a list of choices.



If you need to modify multiple Panorama fields or variables from an embedded code routine you'll need to put all of the data in the output and then parse the results in Panorama. Here's a partial example that puts each data item from a Perl program on a separate line.

```
local contactinfo
contactinfo=perl(|||
    ...
    print name."\n".address."\n".city."\n".state."\n".zip|||)
Contact=firstline(contactinfo)
Address=nthline(contactinfo,2)
City=nthline(contactinfo,3)
State=nthline(contactinfo,4)
Zip=nthline(contactinfo,5)
```

Once the Perl program is finished Panorama splits the returned data into five separate fields. Though this example separates the data items with carriage returns you can use any structure you want. Just make sure the code in your embedded program that generates the output matches your Panorama code that parses it and separates it into individual fields and variables.

Code Embedding Statements and the ScriptResult Variable

Another way to embed code is to use one of the six statements listed below. The most basic versions of these statements use just one parameter, the code to be embedded (other options will be discussed later).

```
applescript code
shellscript code
perl code
ruby code
python code
php code
```

After any of these statements run the global variable **ScriptResult** will contain the value of the data generated by the embedded program. (You don't need to declare this variable with a **global** statement, Panorama does that for you.) This example will get the name of the song currently playing in iTunes and put it into the local Panorama variable **song**.

```
local song
applescript |||tell application "iTunes" to get name of current track|||
song=ScriptResult
```

In some cases you won't care about any data passed back from the embedded program. In that case you can simply use the statement and not worry about what's in **ScriptResult**. For example this program uses AppleScript to reveal the location of the current database (though in reality you would probably simply use Panorama's **revealinfinder** statement to do this):

```
applescript |||
    tell application "Finder"
        activate
        reveal file $«dbname()»$ of folder $«dbpath()»$
    end tell
|||
```

When this program runs the Finder will open a new window displaying the folder that contains the current database. Since opening the folder window is the goal of the program we don't care about the data returned from the embedded program.

Bringing the Embedding Data Output Directly into a Panorama Field or Variable

By adding a second parameter to an embedded code statement you can specify a Panorama field or variable where the result should be placed.

```
applescript code,result
shellscript code,result
perl code,result
ruby code,result
python code,result
php code,result
```

Here is a revised version of the example from the previous section that gets the name of the song currently playing in iTunes and put it into the local Panorama variable `song`.

```
local song
applescript |||tell application "iTunes" to get name of current track|||,song
```

Note that you must choose one method or the other. If you specify a field or variable for the result the result will not be stored in the `ScriptResult` variable.

Advanced Embedding Topics

Now that you've mastered the basics of code embedding the following sections take a look at some of the nitty gritty details.

The Embedded Code Pre Processor

Before passing the code to the appropriate interpreter Panorama runs a simple pre-processor on the code. This pre-processor allows Panorama fields, variables and formulas to be used within the embedded code (see See "[Using Panorama Fields and Variables as Terms in an Embedded Program](#)" on page 755).

The pre-processor is very simple. It simply looks for pairs of `$«` and `»$` tags within the code. If it finds any, it assumes that whatever is between these tags is a Panorama formula, and it evaluates that formula. The pre-processor then replaces both the tags and the formula with the value of the formula expressed as a literal (text or numeric constant) in the target language. For example, suppose the current database has field named `Album` that currently contains the name `Wish You Were Here`. The table below shows how the preprocessor would expand this field in each language.

Language	Original Source	After Preprocessor
AppleScript	<code>set AlbumName to \$«Album»\$</code>	<code>set AlbumName to "Wish You Were Here"</code>
Shell Script	<code>cd \$«Album»\$</code>	<code>cd Wish\ You\ Were\ Here</code>
Perl	<code>\$AlbumName = \$«Album»\$;</code>	<code>\$AlbumName = 'Wish You Were Here';</code>
Ruby	<code>AlbumName = \$«Album»\$;</code>	<code>AlbumName = 'Wish You Were Here';</code>
Python	<code>AlbumName = \$«Album»\$</code>	<code>AlbumName = '''Wish You Were Here'''</code>
PHP	<code>\$AlbumName = \$«Album»\$;</code>	<code>\$AlbumName = 'Wish You Were Here';</code>

If the text contains special characters that need to be escaped the pre-processor will take care of that. For example, suppose the current Album name was `Love Action 'Remix'`. The single quote character requires special handling in several of these languages, as shown below (you'll also notice the special space handling in the shell script example above):

Language	Original Source	After Preprocessor
Perl	<code>\$AlbumName = \$«Album»\$;</code>	<code>\$AlbumName = 'Love Action \'Remix\'';</code>
Ruby	<code>AlbumName = \$«Album»\$;</code>	<code>AlbumName = 'Love Action \'Remix\'';</code>
Python	<code>AlbumName = \$«Album»\$</code>	<code>AlbumName = '''Love Action \'Remix\''''</code>
PHP	<code>\$AlbumName = \$«Album»\$;</code>	<code>\$AlbumName = 'Love Action \'Remix\'';</code>

The bottom line is that you don't need to worry about what characters are in a Panorama field or variable — the pre-processor takes care of any conversion necessary.

The pre-processor also takes care of Panorama fields or variables that contain numbers. For example, suppose the current database has numeric fields named `Width`, `Length` and `Height`, and you have a Ruby function named `shippingrate()` for calculating shipping.

```
myVolume = $«Width»$ * $«Length»$ * $«Height»$;
```

The preprocessor will expand this code like this:

```
myVolume = 12.6 * 4.8 * 9.2;
```


Transferring Dates from Panorama to Embedded Code

The preprocessor does not automatically handle transfer of dates from Panorama to an embedded program. Panorama stores dates as numbers, you'll need to convert this to text with the `datepattern()` function or into separate day, month and year values and then use the appropriate code in the embedded language to convert that text or set of numbers into a date. The example below shows one method (this is Perl, after all, so there are many possible methods) that could be used to transfer a Panorama field named `Date` to a Perl variable named `$startDate`.

```
perl |||
use Time::Local;
$startDate=timelocal(0,0,0,$«dayvalue(Date)»$, $«monthvalue(Date)»$, $«yearvalue(Date)»$);
...
```

Here's a similar method for transferring dates to a Ruby variable:

```
ruby |||
require 'date'
startDate=Date.parse($«datepattern(Date, "mm/dd/yyyy")»$);
...
```

and to a Python variable:

```
python |||
import datetime
startDate=datetime.date($«yearvalue(Date)»$, $«monthvalue(Date)»$, $«dayvalue(Date)»$)
...
```

and finally a method to do this in PHP:

```
php |||
$startDate = strtotime($«datepattern(Date, "mm/dd/yyyy")»$);
...
```

Using Panorama Formulas “Bare” within an Embedded Program

You've already seen how to use the `$«` and `»$` tags to insert a Panorama formula as a term in the embedded program. The preprocessor also supports alternate `^«` and `»^` tags that insert the value of the formula “bare” into the embedded code. By “bare” we mean that the value is inserted exactly as-is into the embedded program, with no quotes or escaping of special characters. We can think of several uses for this. First, you could use this to insert complete statements or even entire blocks of code into an embedded program. This example assumes that your database contains three fields named `File`, `Folder` and `Action`. The `Action` field must contain either `open` or `reveal`. Depending on which of these words the `Action` field contains the procedure will either open the specified file or open the Finder window containing the file.

```
if Action match "open" or Action match "reveal"
  applescript |||
  tell application "Finder"
    activate
    ^«Action»^ file $«File»$ of folder $«Folder»$
  end tell
  |||
else
  message "Action must be open or reveal."
endif
```

A bare formula can also be useful for initializing lists, hashes and arrays. Here is an example that converts a Panorama carriage return delimited array (a list of field names in the current database) into a list in a Perl variable:

```
perl |||
@fieldnames = (^<replace(arrayfilter(dbinfo("fields",""),cr()),
    {perlconstant(import())}),cr()," , ")>^)
```

When the pre-processor gets done with this code it will look something like this:

```
@fieldnames = ('Name' , 'Address' , 'City' , 'State' , 'Zip')
```

Bare formulas can also be used with special quoting in the embedded language. This example assigns the contents of the Panorama field **Memo** to the Perl variable `$memoText`, but converts the text to upper case first.

```
perl |||
$memoText=<<"FOO";
\U^<Memo>^\E
FOO
...
```

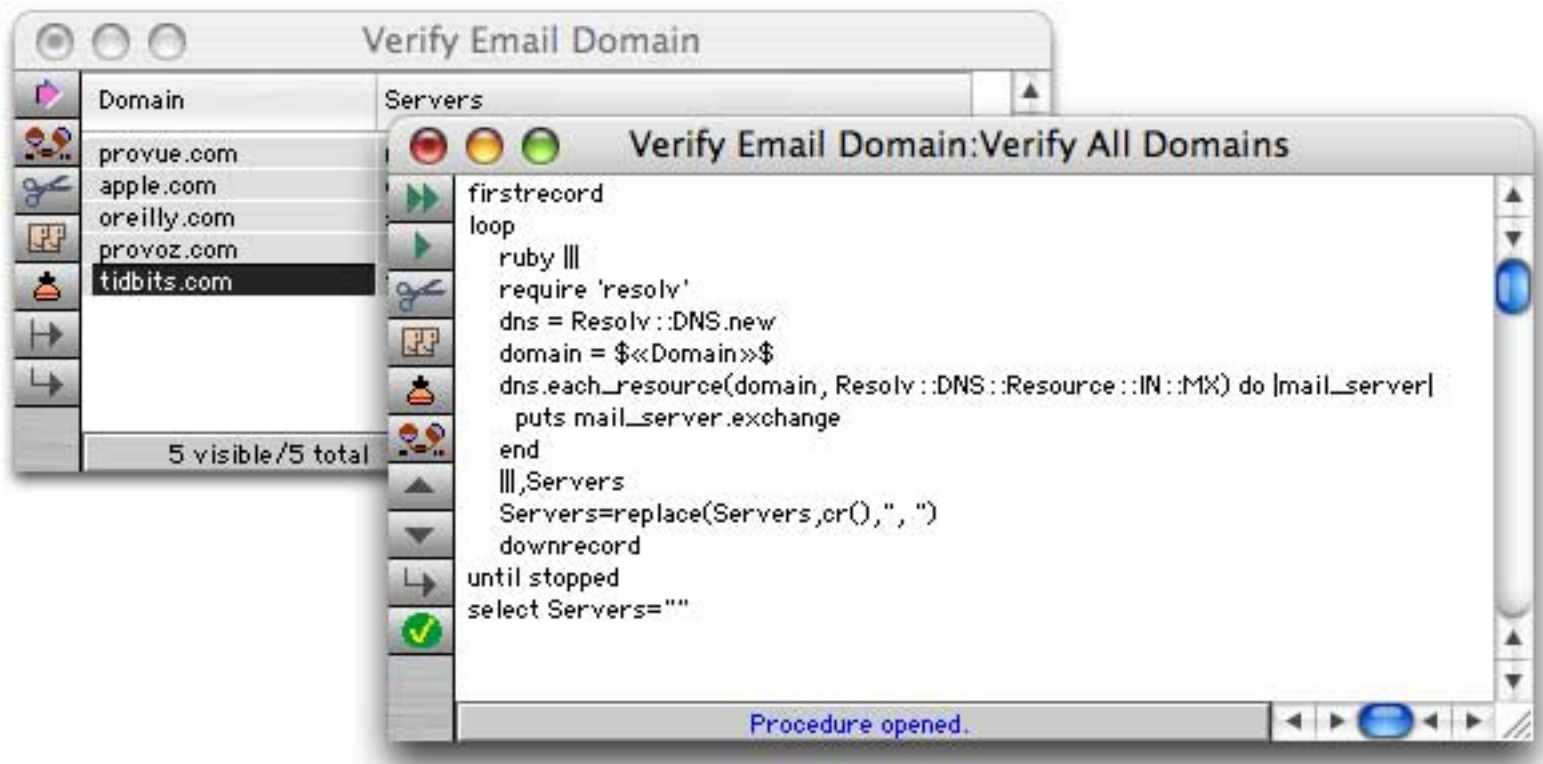
Generating Constant Values. In the previous section the `perlconstant()` function was used to generate a Perl literal value within the `arrayfilter()` function. This function takes a single parameter and converts that parameter into a Perl literal value suitable for insertion into a Perl program. If the parameter contains text then quotes are added and any special characters are embedded. Panorama actually has six such functions, one for each of the embeddable languages.

```
applescriptconstant(value)
unixshellstring(value)
perlconstant(value)
rubyconstant(value)
pythonconstant(value)
phpconstant(value)
```

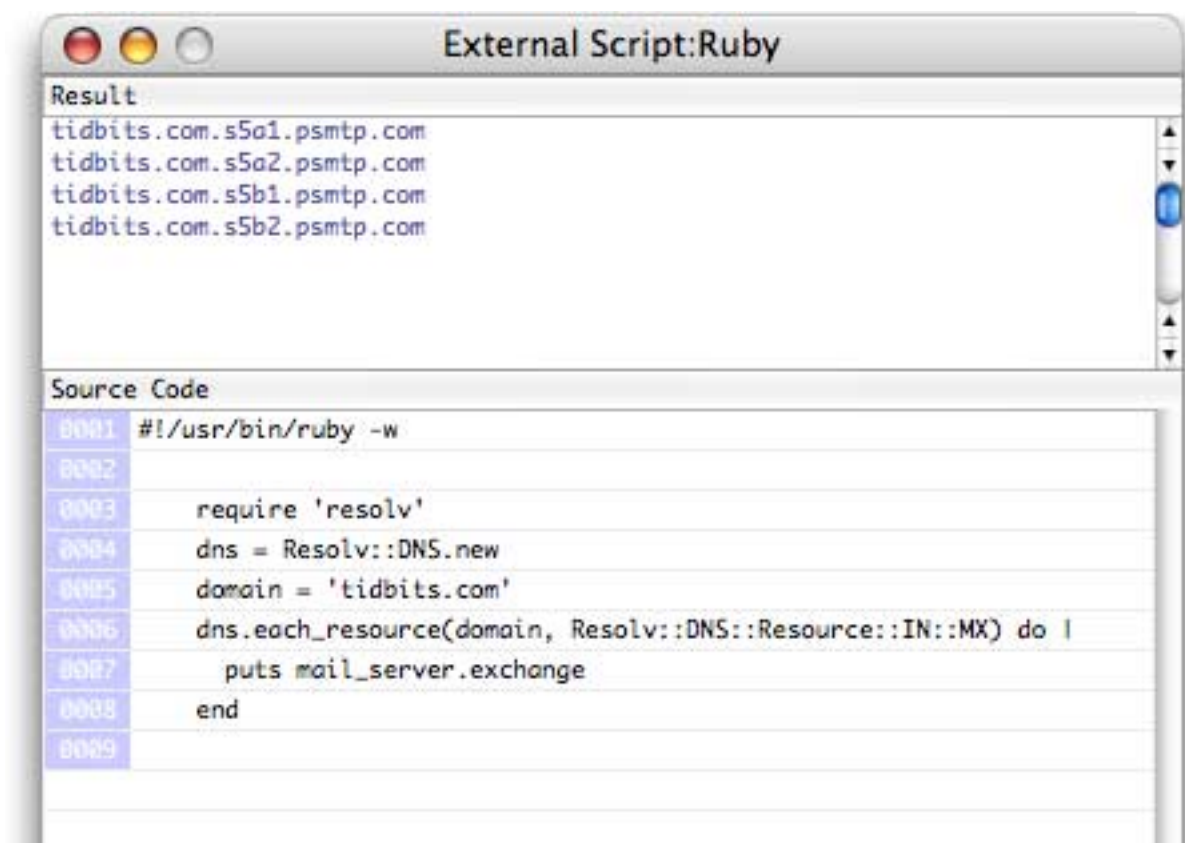
Each of these function generates the appropriate quotes and escaped special characters for the corresponding language. (Note: You shouldn't ever need to use these function unless you are using bare Panorama formulas within your embedded programs. The pre-processor already uses these functions for you when you insert formulas with the normal `$<` and `>$` tags.

The External Script Wizard

The **External Script** wizard (in the **Developer Tools** submenu of the **Wizard** menu) can help you debug embedded programs. The wizard displays the last embedded program that has been run, showing both the output of the pre-processor (see “[The Embedded Code Pre Processor](#)” on page 760) and the result produced by the program (usually standard output). For example, consider this database that contains E-mail domains along with an embedded Ruby program for verifying domains.



When the procedure runs it will loop from the top to the bottom of the database, executing the embedded Ruby code once for each record in the database. To see the results of the final time the Ruby code was executed open the **External Script** wizard:



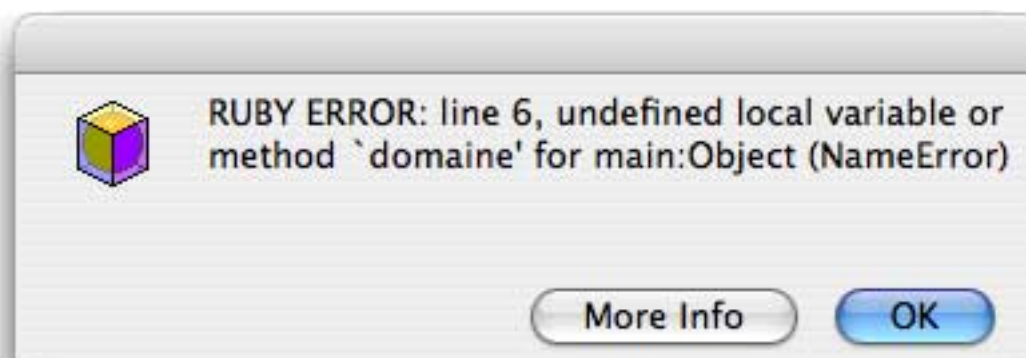
The top section of the wizard displays the results produced by the Ruby code. In this case [tidbits.com](#) is a valid domain name and has four MX records. The bottom section of the wizard shows the exact source code that was submitted to the Ruby interpreter. Look at line 5 — you can see that the pre-processor replaced `$<<Domain>>$` with `'tidbits.com'`.

Dealing With Errors in Embedded Programs

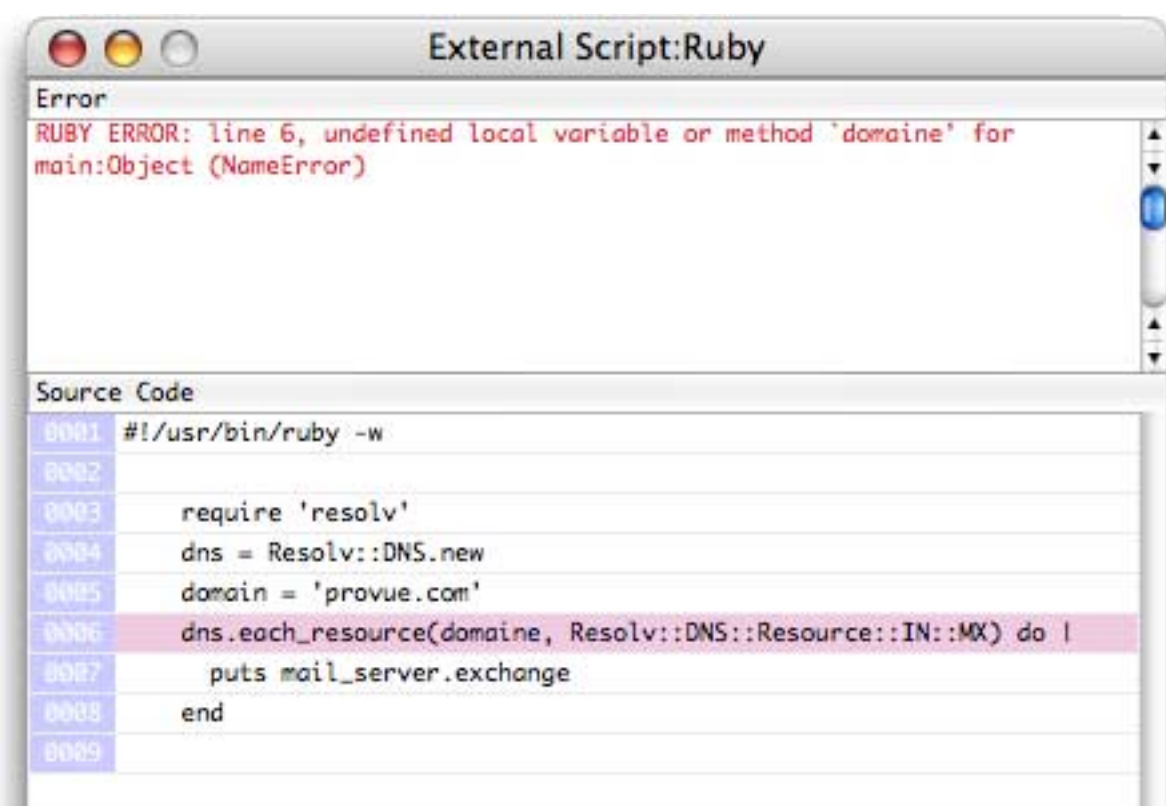
The **External Script** wizard is especially handy if there is a typo or error in an embedded program. For example, suppose you had accidentally typed an extra **e** at the end of **domain**.



When you run this procedure an error message will appear:



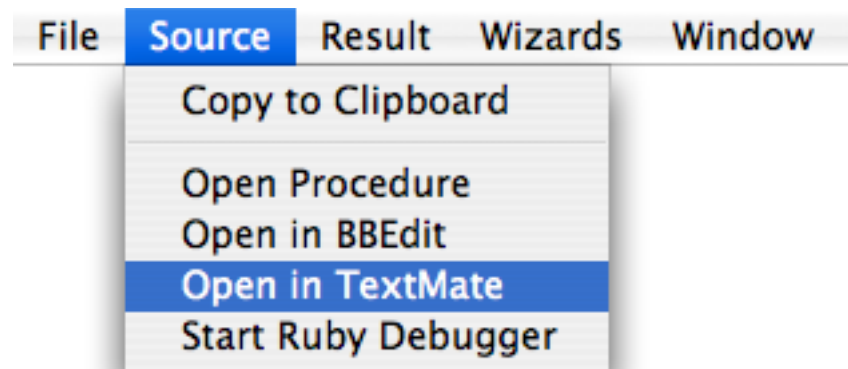
Too cryptic for you? Press the **More Info** button. Both the **Error Detail** and **External Script** wizards will open automatically. The **Error Detail** wizard (see also) will tell you where in your Panorama code the external script is located (sometimes this may not be obvious if you have multiple levels of subroutines). The **External Script** wizard displays the embedded program, the error message, and (if possible) highlights the line containing the error.



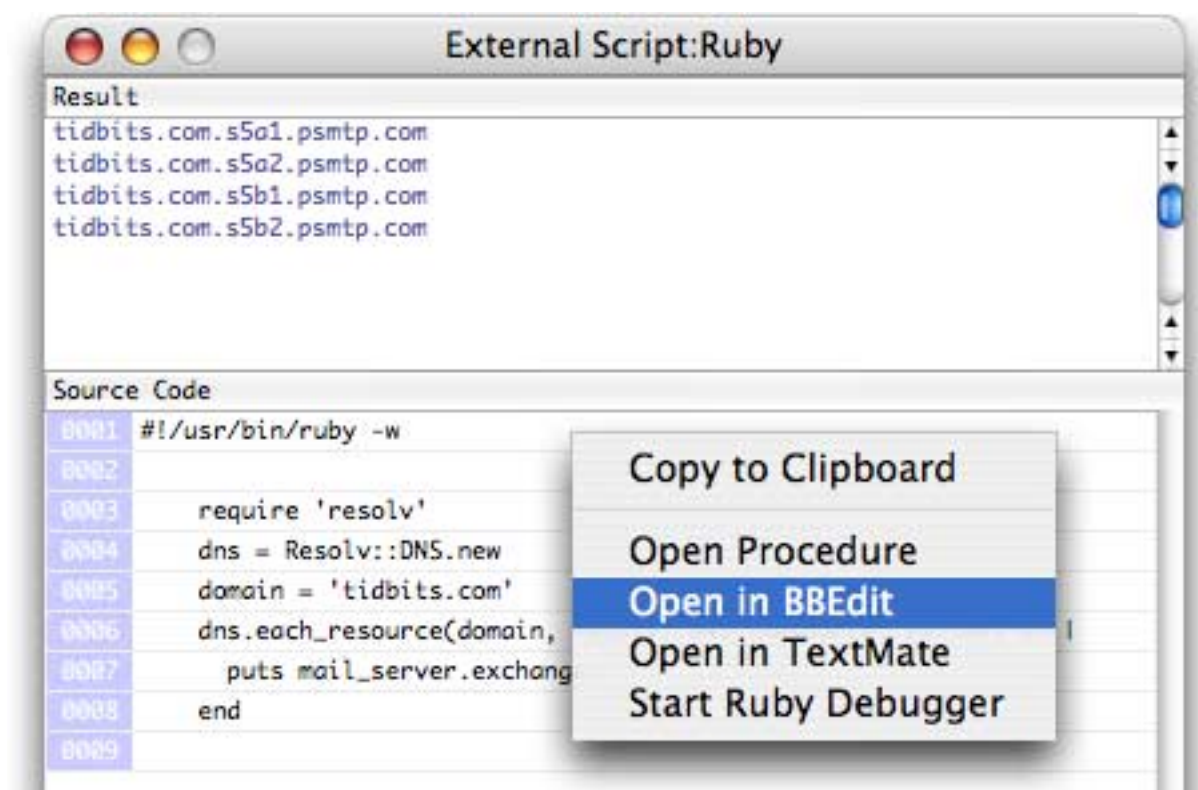
To fix this problem you'll need to go back to the original Panorama procedure that contains the embedded program. If the procedure isn't already open (or if it is hidden behind other windows) you can open it by choosing **Open Procedure** from the **Source** menu.

Working with External Editors

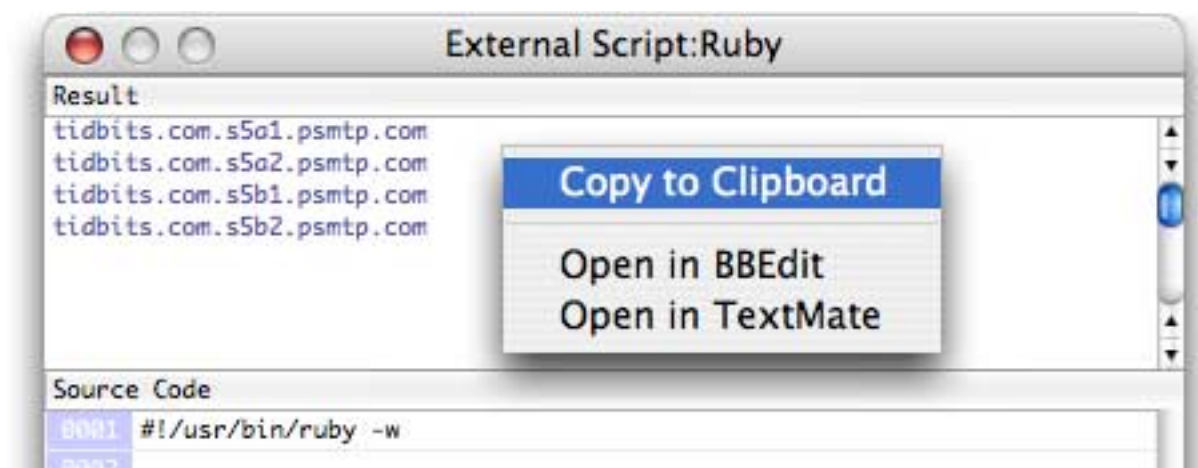
Sometimes you may want to do some testing or other work on an embedded program in an external editor. To do this simply choose the editor from the **Source** menu.



The wizard will save the embedded program in a temporary file and open the requested editing program. You can also choose from this menu simply by clicking on the source code — a pop-up menu appears.



You can also work with the results in an external editor. Simply click on the results for a pop-up menu or chose from the **Result** menu.



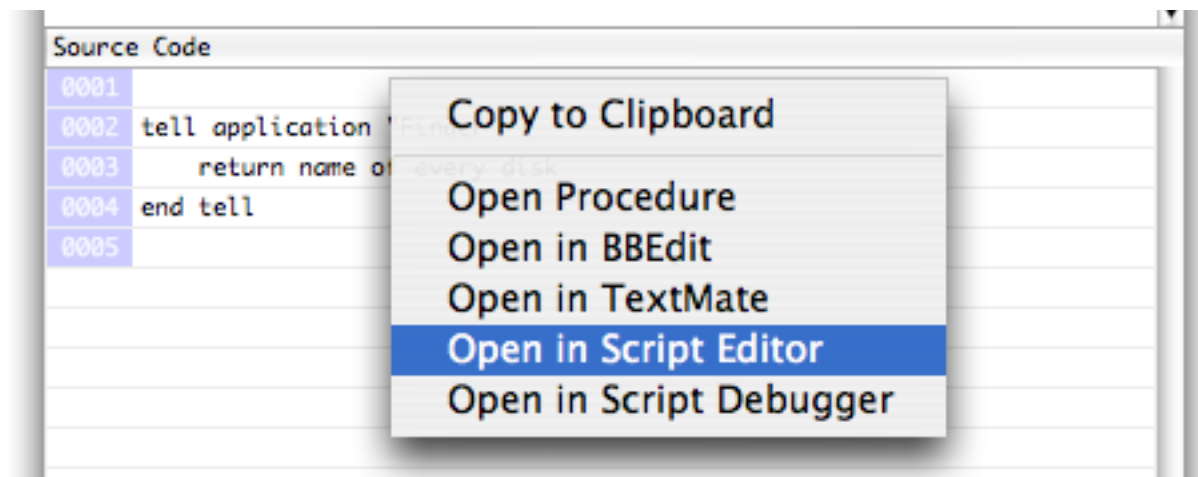
When you work on the source code in an external editor there is no automatic way to get your changes back into Panorama. (Usually you wouldn't want to anyway, since this source code has already been run through the pre-processor.) You can, of course, use copy and paste to bring some or all of the program back into your Panorama procedure.

Working with External Debuggers

For some languages the **External Script** wizard supports the use of external debuggers. At the time this is being written debugger integration is available for AppleScript, shell scripts, Perl and Ruby. For each of these languages the debugger is accessible from the **Source** menu, either in the menu bar or as a pop-up menu by clicking on the source code.

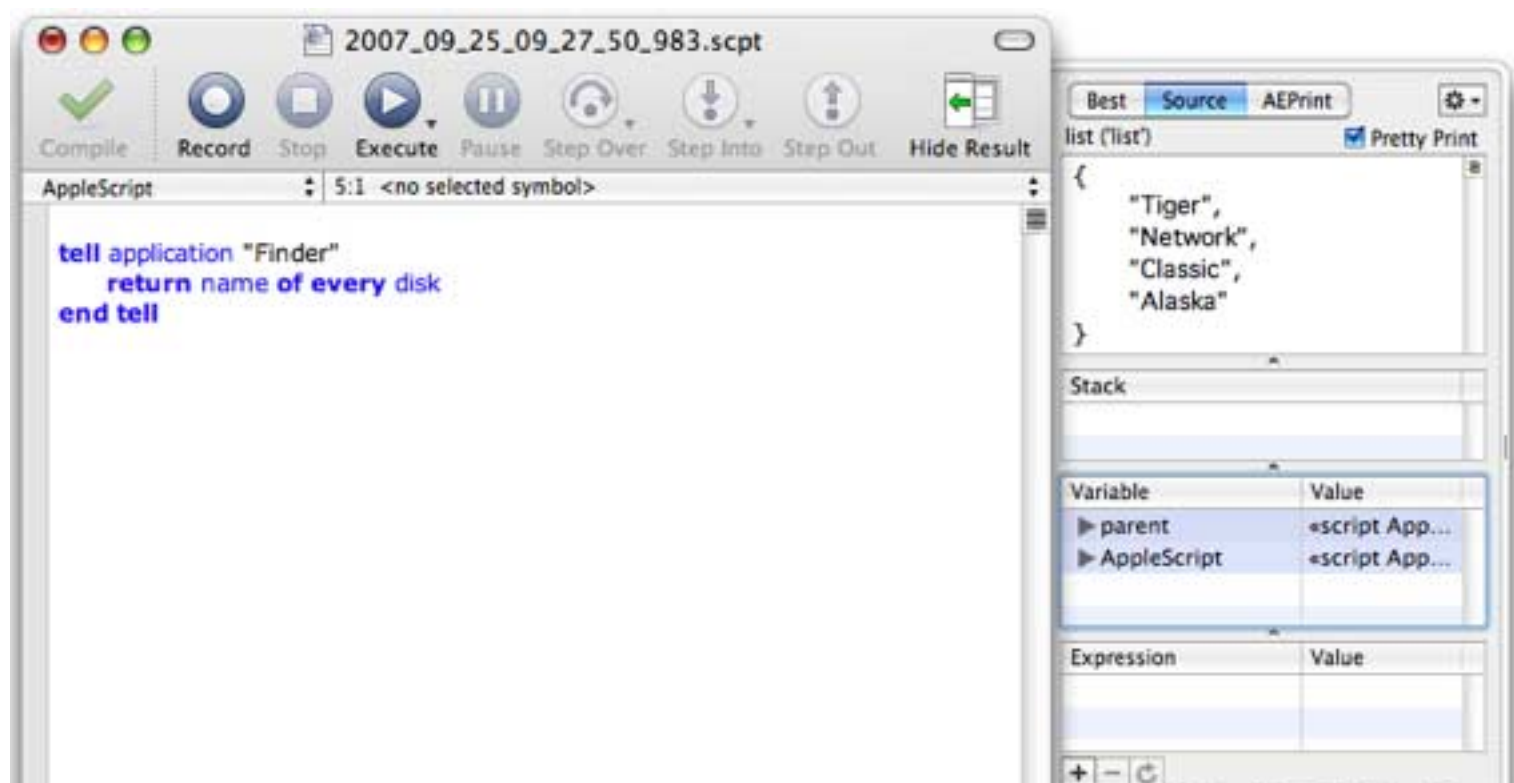
AppleScript Debuggers

The **Source** menu contains two programs that can help with debugging, **Script Editor** and **Script Debugger**.



Script Editor is included with every Macintosh system. It's not really a debugger but it does allow you to make changes and try them out.

Script Debugger is a commercial program for debugging AppleScripts. It's from a company called *Late Night Software* and if you are planning on doing any serious work with AppleScript we highly recommend that you check out this program. **Script Debugger** isn't cheap, but the investment will pay for itself quickly if you are writing scripts of any complexity at all. Its ability to debug, examine variables, and explore the object model have turned AppleScript projects we work on here at ProVUE Development from a frustrating hit or miss experience into a smooth productive workflow. Working with **Script Debugger** is like turning on the light in a dark room. When you select **Script Debugger** from the **Source** menu the wizard saves a temporary file containing the script and fires up the software, as shown here:

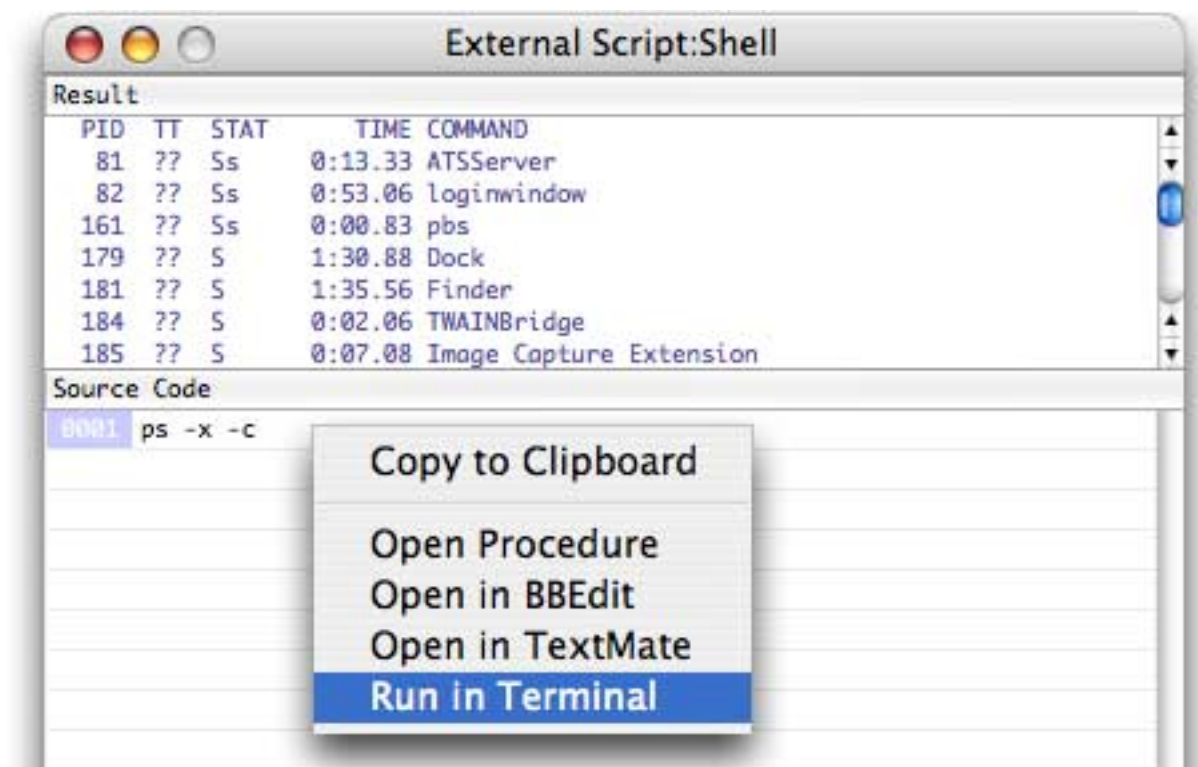


We don't have any connection with *Late Night Software*, but we do recognize great tools when we use them, so we wanted to pass along the secret to you. You can find out more at:

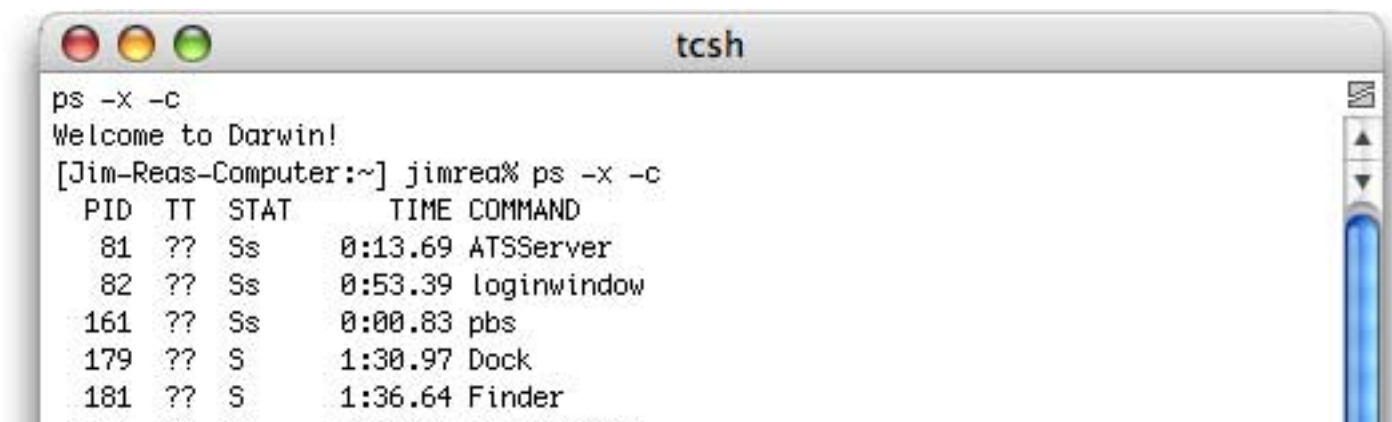
<http://www.latenightsw.com>

Shell Scripts

There's no actual debugger for shell scripts, but you can run them in Apple's Terminal program:

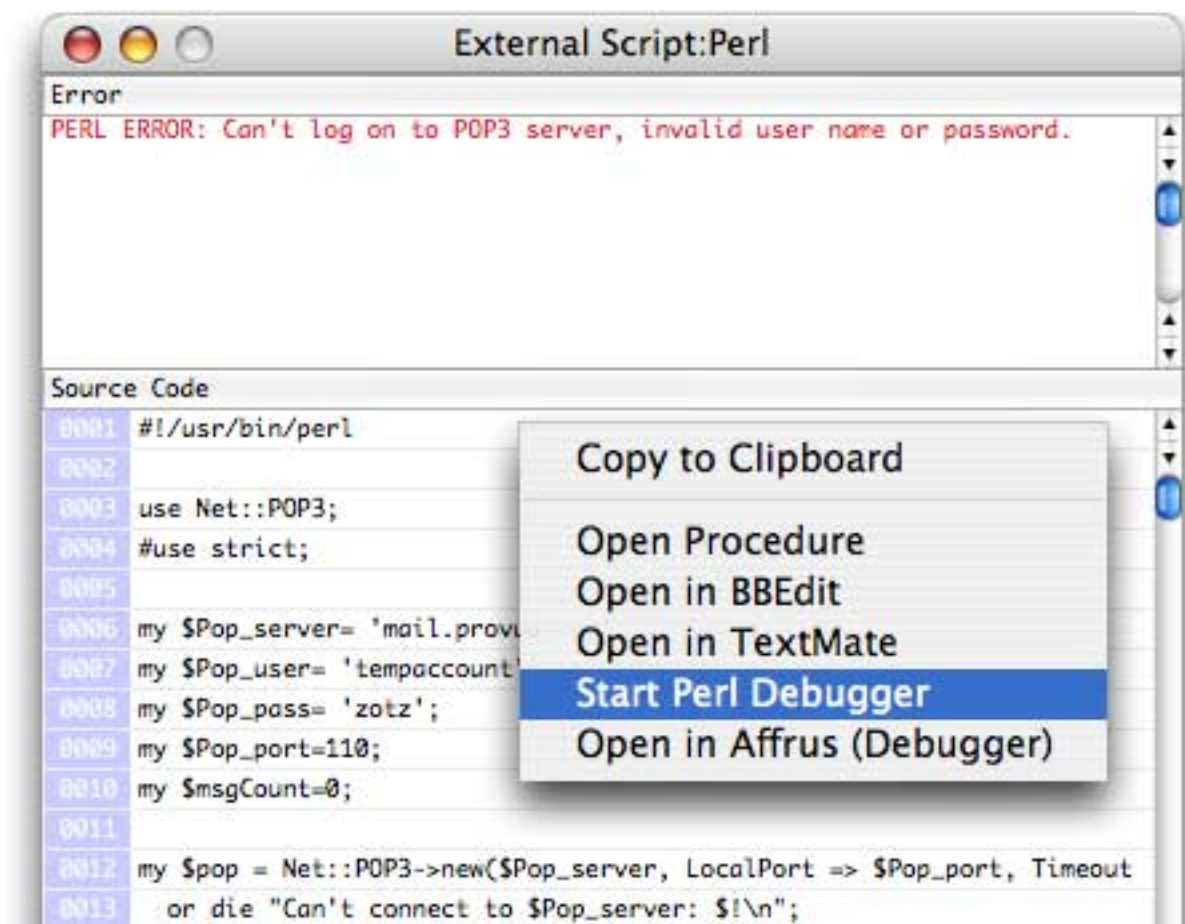


Choosing this command opens a new window in the Terminal program and runs the command.

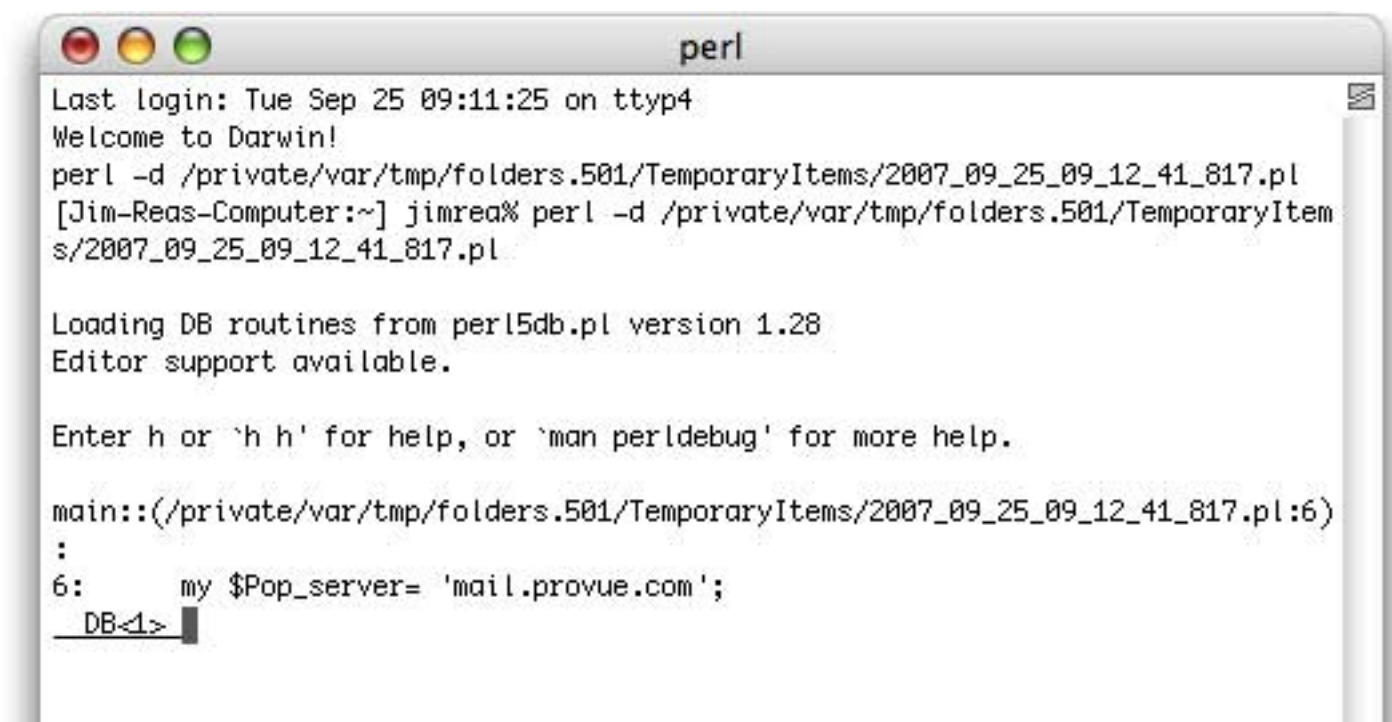


Perl Debuggers

The wizard is integrated with two Perl debuggers.



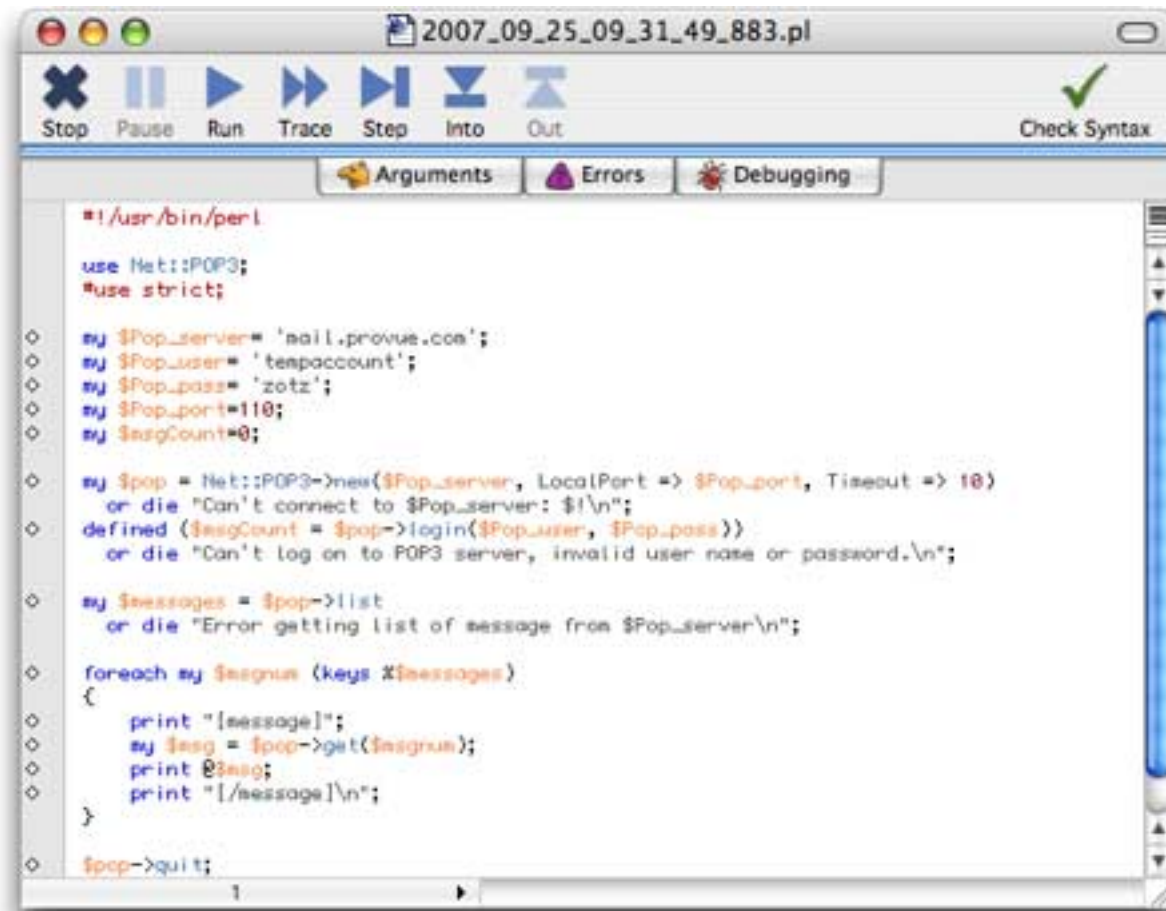
The first option is the standard **Perl Debugger**, which is included free with Perl. When you choose this option Panorama automatically opens a new Terminal window and launches the debugger with this embedded program.



At this point you can type in various commands to single step, set breakpoints, examine variables, etc. Everything is done by typing in commands. To learn more about how to use this debugger see the **Programming Perl** book mentioned earlier in this chapter (see "[Perl](#)" on page 750) or online at these urls.

URL	Description
http://perldoc.perl.org/perldebug.html	Perl Debugger Reference Manual
http://perldoc.perl.org/perldebtut.html	Perl Debugger Tutorial

The second option is **Affrus**, another commercial program we like from *Late Night Software*. Affrus is an integrated visual Perl editing and debugging environment for Mac OS X. The debugger features step-wise execution, breakpoints, tracing, and expression evaluation as well as stack frame tracing with full access to locally scoped (my) variables. The debugger also displays all Perl registers and variables from any Package. When you select Affrus from the Source menu the wizard saves a temporary file containing the script (after pre-processing) and fires up Affrus, as shown here:

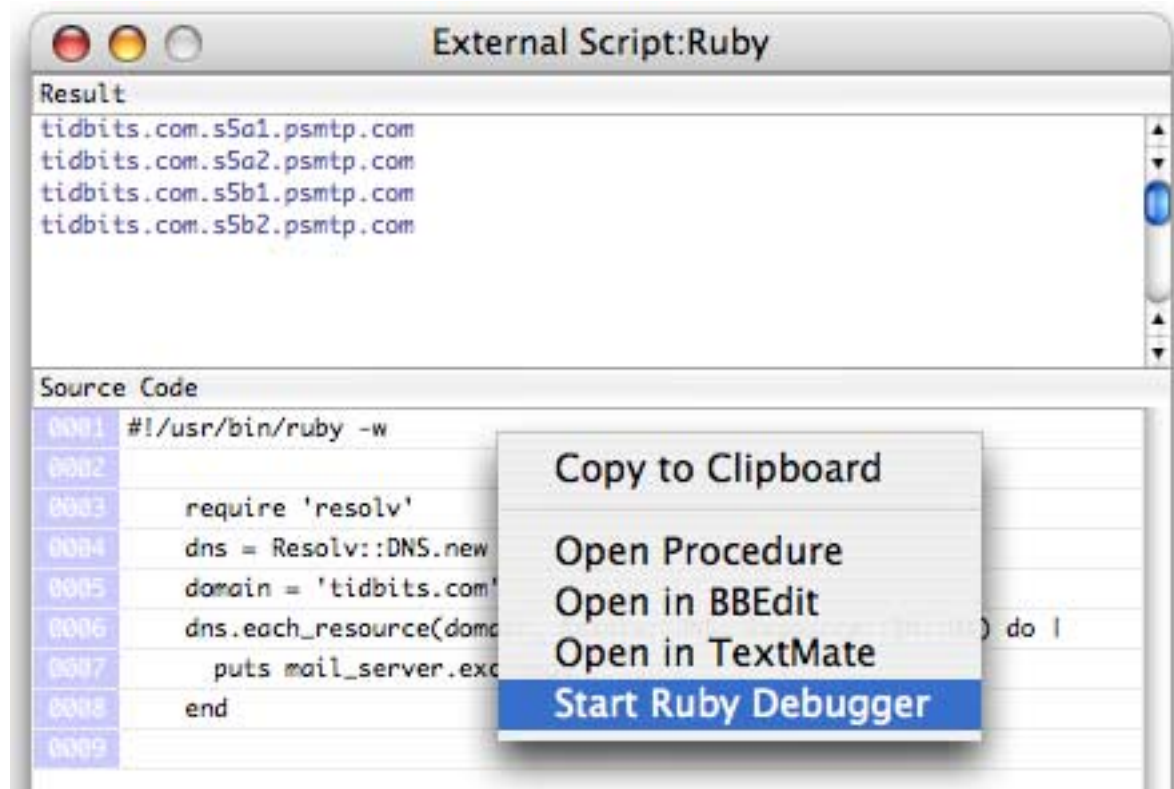


You can learn more about Affrus at:

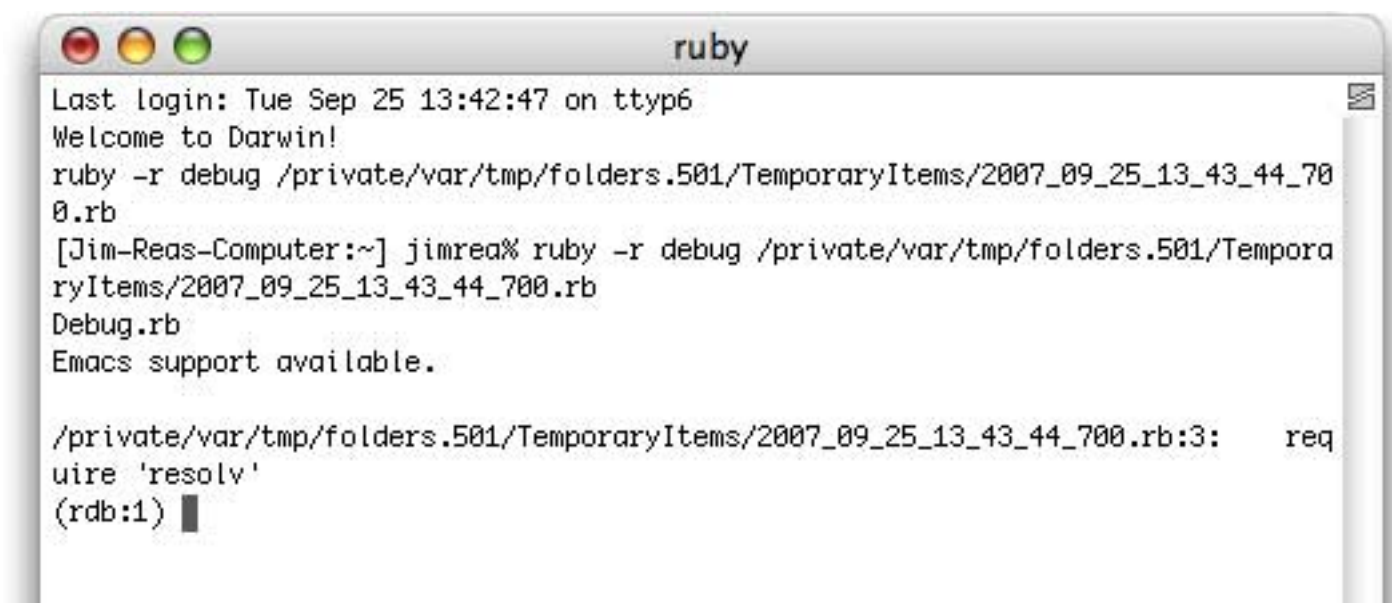
<http://www.latenightsw.com>

Ruby Debuggers

To launch the standard Ruby debugger choose **Start Ruby Debugger** from the **Source** menu. This is the standard command-line debugger that comes free with Ruby.



When you choose this option Panorama automatically opens a new Terminal window and launches the debugger with your embedded Ruby program.



At this point you can type in various commands to single step, set breakpoints, examine variables, etc. Everything is done by typing in commands (you can get a list of the available commands by typing **help** followed by carriage return). To learn more about how to use this debugger see the **Programming Ruby** book mentioned earlier in this chapter (see "**Ruby**" on page 751). At the time this is being written there is no "official" documentation online for the ruby debugger, but the debugging chapter from the 1st edition of the **Programming Ruby** book is available at:

<http://www.ruby-doc.org/docs/ProgrammingRuby/html/trouble.html>

Special Embedding Options

In addition to the options already described Panorama allows you to specify the amount of time an embedded program is allowed to run and the temporary folder used to run an embedded program. You can also specify that shell scripts must run with temporary root privileges.

Specifying the Maximum Embedded Program Runtime

Panorama will normally wait up to 60 seconds for an embedded program to complete. If the embedded program does not complete after 60 seconds Panorama will terminate the program and return an error. If you want a longer (or shorter) maximum time you can specify the maximum runtime. If you are using the `applescript` or `shellscript` statements the maximum runtime is the third parameter, for `perl`, `ruby`, `python` and `php` it is the fourth parameter.

```
applescript code,result,timeout
shellscript code,result,timeout
perl code,result,folder,timeout
ruby code,result,folder,timeout
python code,result,folder,timeout
php code,result,folder,timeout
```

For example, suppose you want to embed a complex Ruby program that may take several minutes to run. This example will allow the code to run up to 10 minutes (600 seconds).

```
local rubyResult
ruby |||
...
...
... very long ruby program
...
...
|||,rubyResult,"",600
```

You can also specify maximum script runtime in advance with the `scripttimeout` statement.

```
local rubyResult
scripttimeout 600
ruby |||
...
...
... very long ruby program
...
...
|||,rubyResult
```

The `scripttimeout` statement also allows you to change the timeout for the embedded code functions (`applescript`(, `shellscript`(, `perl`(, etc.)

Running Shell Scripts with Temporary Root Privileges (SUDO)

Embedded code usually inherits whatever privileges the current user has. If you need to run a shell script with elevated privileges you must supply a password parameter to the `shellscript` statement (the fourth parameter).

```
shellscript code,result,timeout,password
```

However, you don't actually have to supply the actual password here. If the password is blank, the `shellscript` statement will prompt for the password when it runs:

```
local scriptResult,systemPassword
systemPassword=""
shellscript |||diskutil repairPermissions /|||,scriptResult,60*20,systemPassword
```

You can also build the password into the procedure itself, like this:

```
local scriptResult,systemPassword
systemPassword="zorax72"
shellscript |||diskutil repairPermissions /|||,scriptResult,60*20,systemPassword
```

Building the password into the procedure isn't usually a great idea, however. You should carefully consider the security implications before using this technique.

Specifying the Embedded Code Folder

To execute Perl, Ruby Python or PHP code Panorama creates a temporary file containing the code and uses the command line to invoke the appropriate interpreter. This temporary file is usually created in a special folder designated by the operating system for temporary files. You can use the `folder` parameter to specify a different folder for the temporary code file. This example puts the temporary file in a folder named **Perl Code**, which is in a subfolder of the folder containing the current database.

```
perl |||... perl code ...|||,scriptResult,dbsubfolder("Perl Code")
```

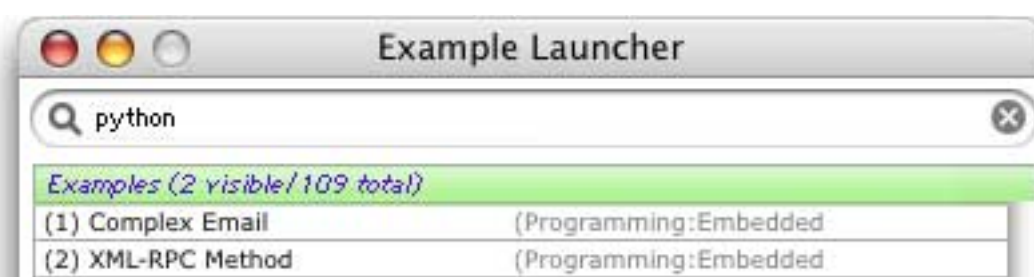
You can put the temporary file anywhere you want. It will be erased once it the interpreter is finished with it.

Real World Embedded Code Examples

Often the best way to learn a new programming technique is to study a working example. The Panorama example folder includes a number of sample databases that illustrate programming in each of the six available embedded languages. To access these databases simply open the **Example Launcher** wizard (in the **Demo** submenu of the **Wizard** menu) and search for **embed**, as shown below. (If you don't see these files then you need to download the deluxe version of Panorama. There's no extra charge for the deluxe version, but it is a larger download.)



If you only want to see the example for a particular language simply type in that language, like this:



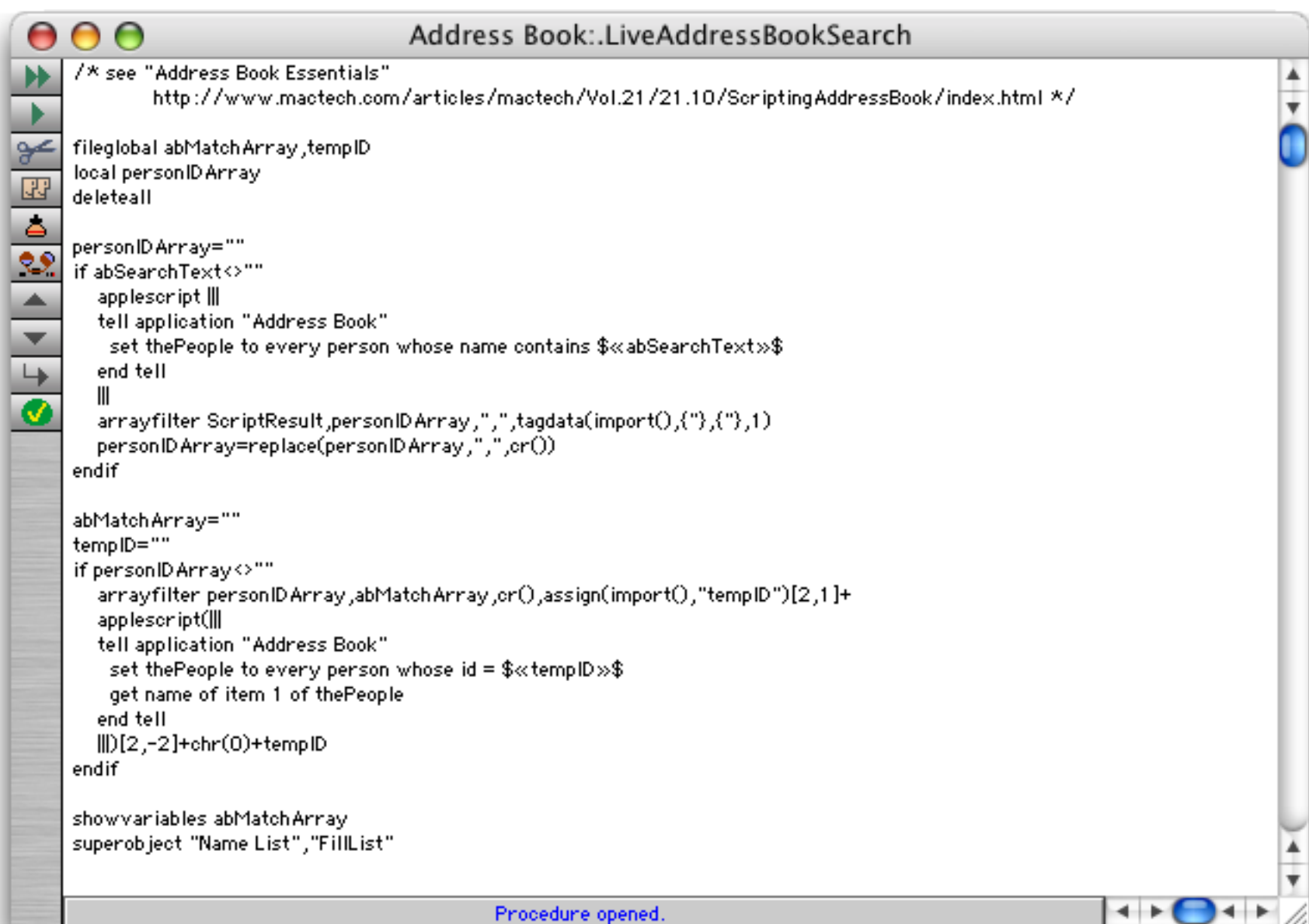
The following sections will briefly discuss some of these example files. (Note: These examples are designed for teaching purposes, and are not necessarily "production" quality with all the bells, whistles, error checking etc. We also can't guarantee that these examples will work on your computer, or provide support if they don't. On a more positive note you should keep in mind also that these examples only scratch the surface of the possibilities enabled by embedding these languages in your Panorama code.)

AppleScript — Address Book Search

This database implements a “Live Clairvoyance” style search, but instead of searching a database it searches Apple’s **Address Book** application.



Each time a key is pressed the Panorama program shown below is run:



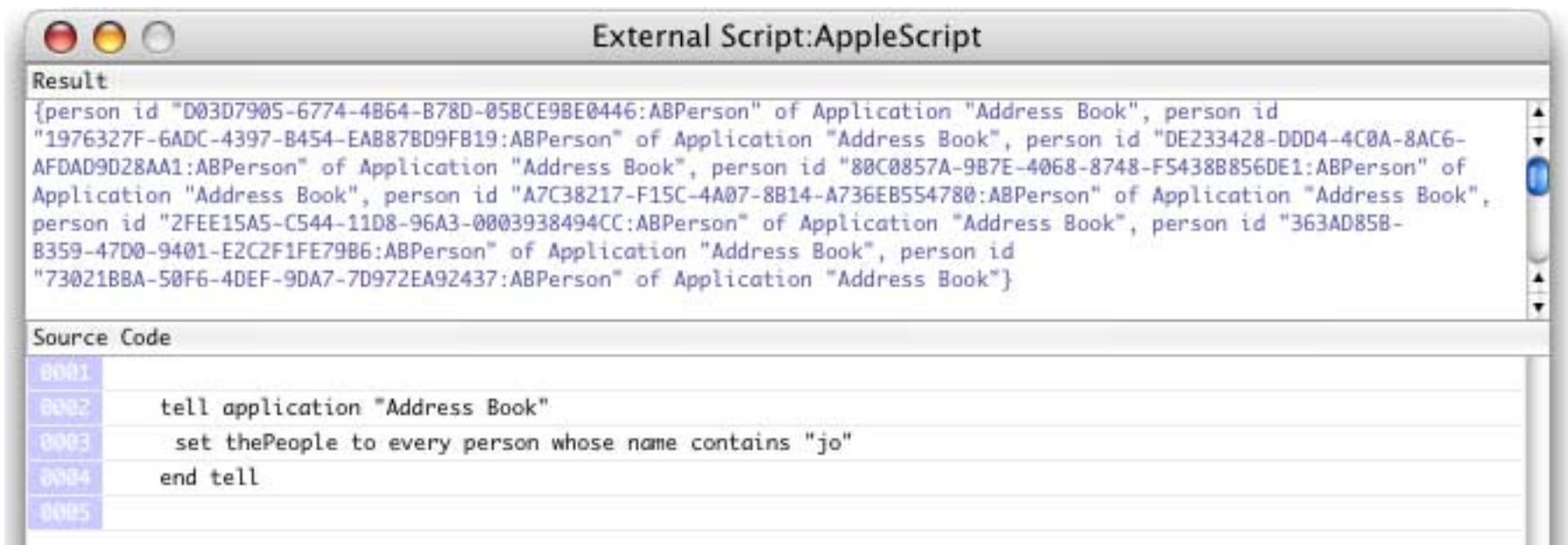
The program actually contains two separate AppleScripts. The first gets a list of people that contain the text that has been entered (which is in a Panorama variable named `abSearchText`).

```

tell application "Address Book"
  set thePeople to every person whose name contains $«abSearchText»$
end tell

```

This script returns a list of internal ID's for the matching people:



As you can see, these internal ID's are pretty much unintelligible on their own. The second half of the program turns these ID's into a carriage return delimited array, then uses the arrayfilter statement to repeatedly call another AppleScript to get the names associated with these ID's.

```

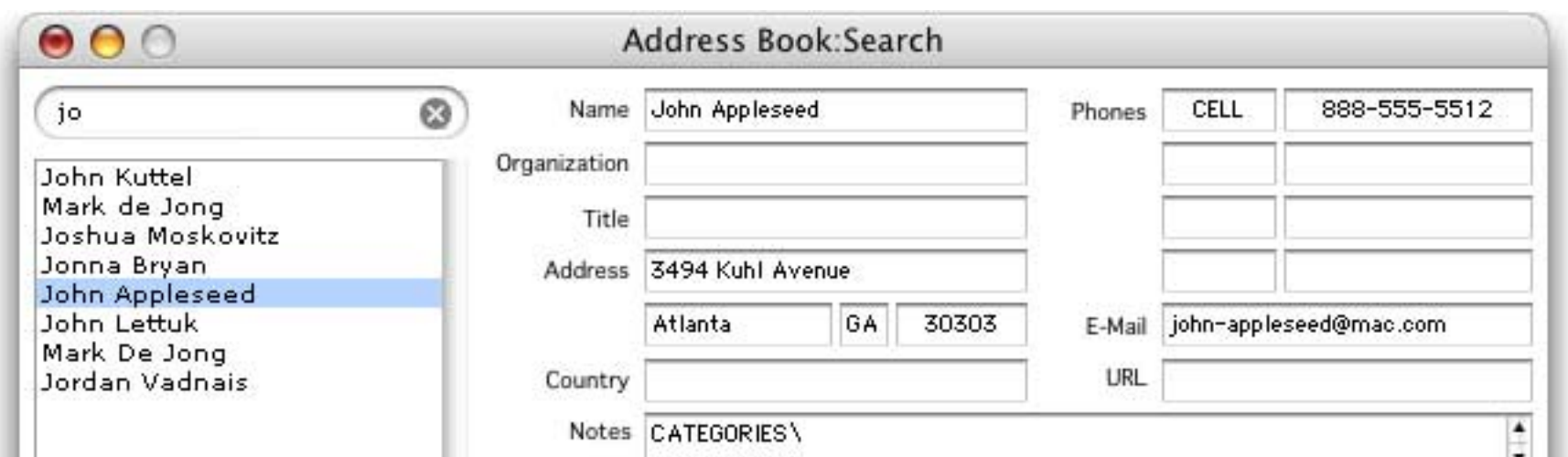
arrayfilter personIDArray,abMatchArray,cr(),assign(import(),"tempID")[2,1]+
applescript(|||
    tell application "Address Book"
    set thePeople to every person whose id = $«tempID»$
    get name of item 1 of thePeople
end tell
|||)[2,-2]+chr(0)+tempID

```

There is a bit of a trick in the code above. Panorama's `import()` and `seq()` functions don't work inside embedded code. In this case the `import()` function contains the ID's we need to feed back to AppleScript, so we really need that value in the embedded code. To get around this the code uses the `assign()` function to temporarily assign the ID to a fileglobal named `tempID`. The `tempID` variable is then embedded in the AppleScript. (You could eliminate the need for this trick by using a regular loop instead of `arrayfilter`, but `arrayfilter` is quite a bit faster.)

The end of this code segment shows another lesser known trick. Panorama's List Object will not display any text in a line that comes after a `chr(0)` byte. This allows us to embed the internal ID in the list without displaying it. We'll use this internal ID in a moment.

Once you've found the person you want you can click on their name to see the detailed information for that person:



Clicking on a person's name in the list triggers this procedure:



This starts by using an AppleScript to get all of the detailed information for this person:

```

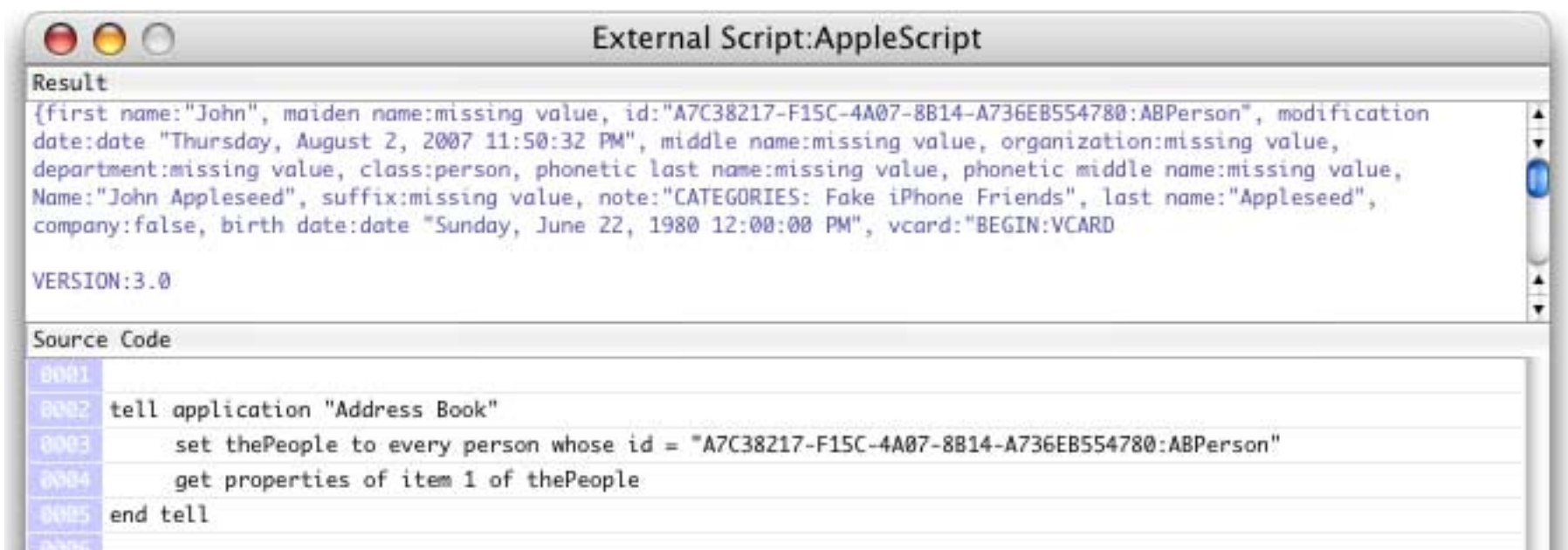
tell application "Address Book"
    set thePeople to every person whose id = $«array(abMatchItem,2,chr(0))»$
    get properties of item 1 of thePeople
end tell

```

The variable `abMatchItem` contains the choice the user clicked on. Remember that the internal ID for each person is embedded in each choice after a `chr(0)` byte? The `array()` function extracts this ID so that it can be submitted to Address Book through the script.

```
array(abMatchItem,2,chr(0))
```

The result of this script looks like this:

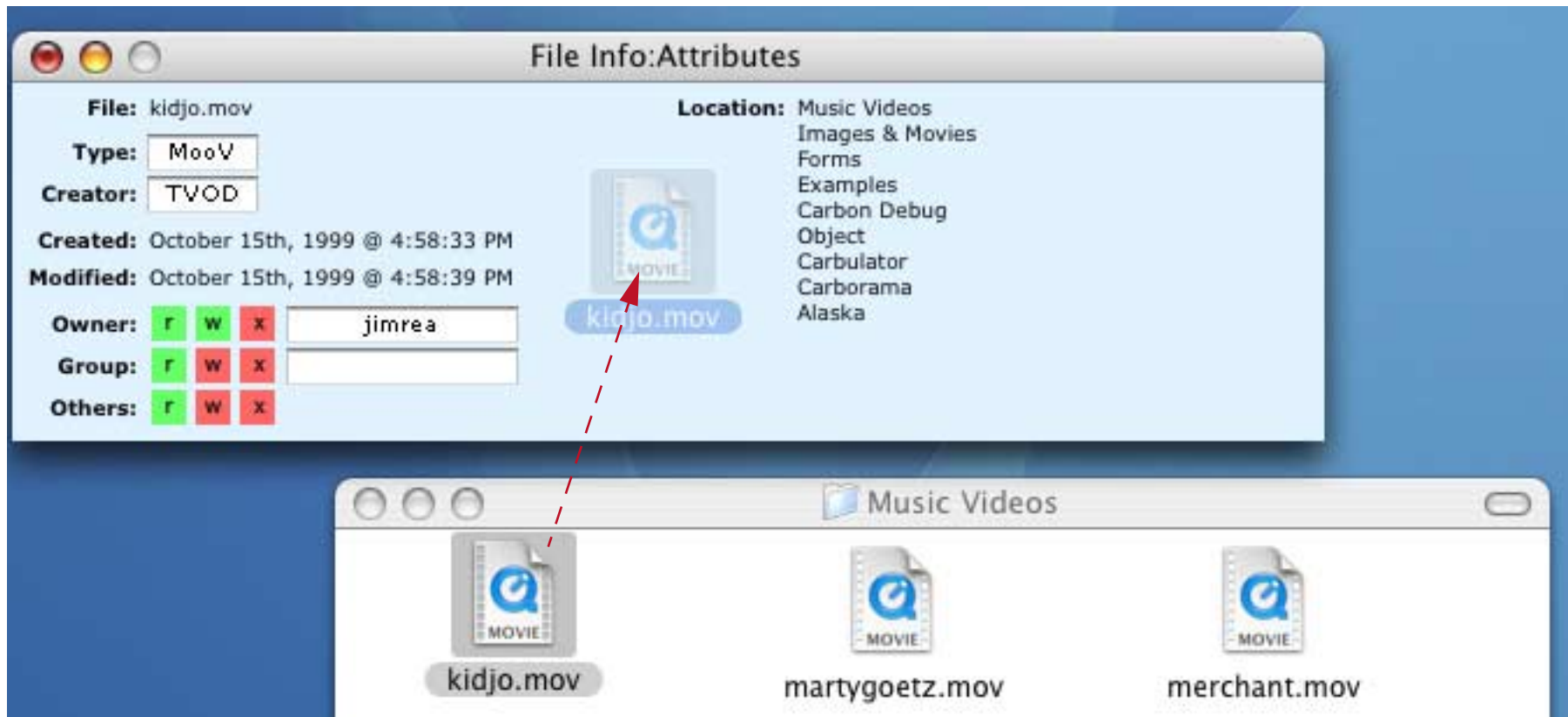


You can't see all of it in this screen shot, but the result contains a VCard with the information we want to display. Since Panorama knows how to import VCards we simply import this information into the database to display it.

This simple example doesn't allow us to do anything beyond searching and displaying Address Book entries, but it could be extended to allow you to modify entries or to transfer the information to other databases.

Shell Script — File Info

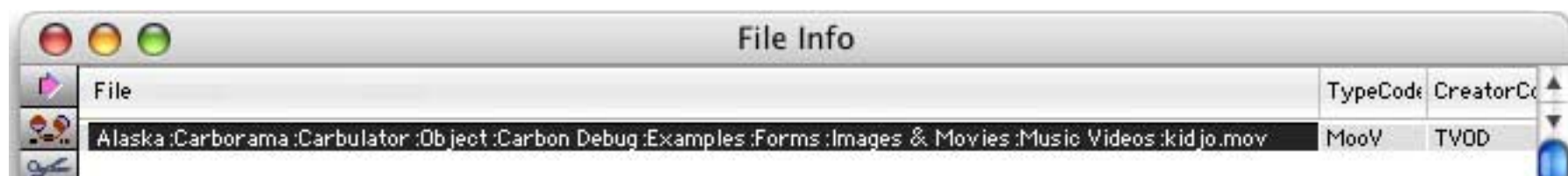
This example database uses shell scripts to examine and change file information. You can drag any file from the Finder onto the database and it will display the attributes of the file, including type, creator, permissions and owners.



When a file is dropped on the database the procedure below is executed:



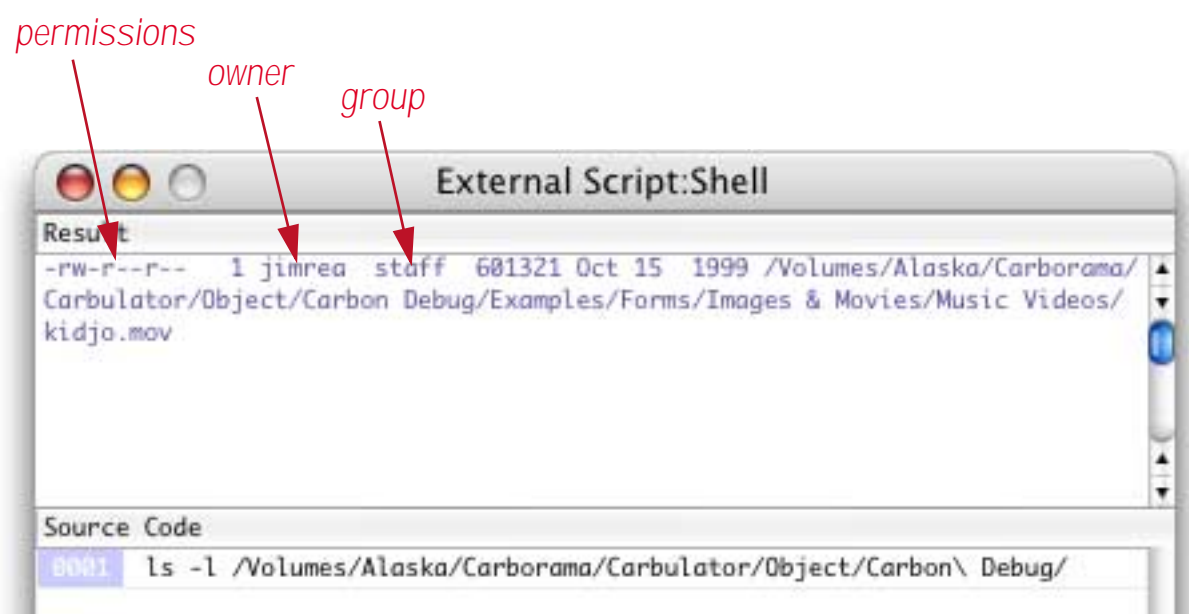
For the purpose of this chapter we are primarily interested in the second half of this procedure, which embeds a shell script with the `ls` command to find out the owner and permissions for the dropped file. At this point the filename is stored in the field File in standard HFS format:



The embedded code uses the `unixshellpath()` function to convert this path and filename into UNIX format.

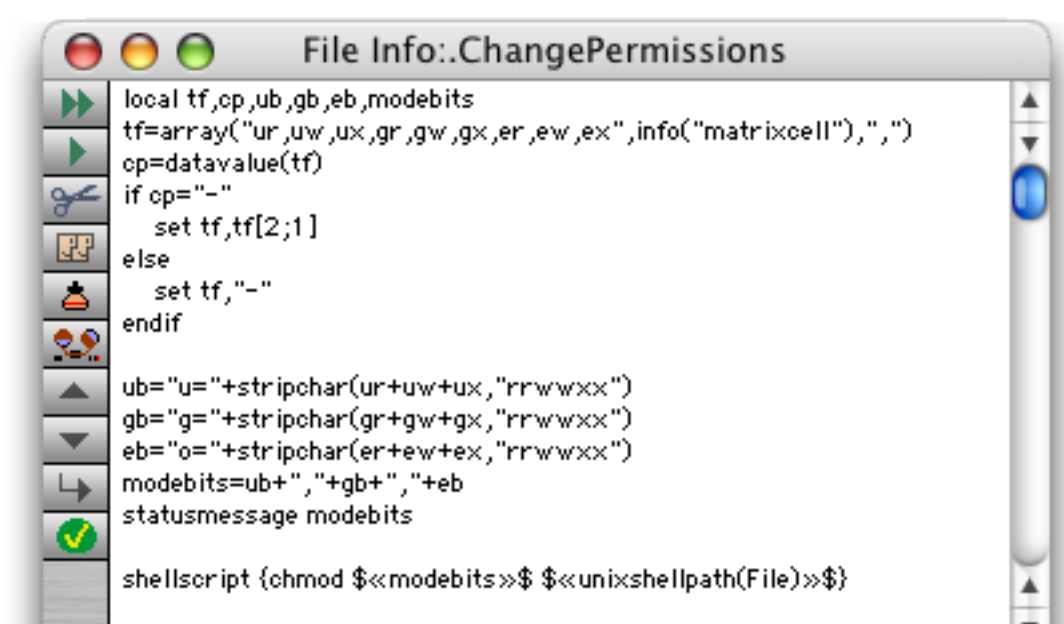
```
shellscript ||| ls -l $«unixshellpath(File)»$ |||
```

The result will look something like this:



The remainder of the `.GetAttributes` procedure takes these results and parses them into separate Panorama fields for display.

In addition to displaying attributes this procedure also allows you to change them. Clicking on one of the nine permission boxes triggers this procedure:



The first half of the code toggles the appropriate field in the database. The second half calculates the mode argument for the unix `chmod` command. The final line runs the `chmod` command to actually change the permissions of the file itself. Try this yourself and use the **External Script** wizard to see the command line being generated.

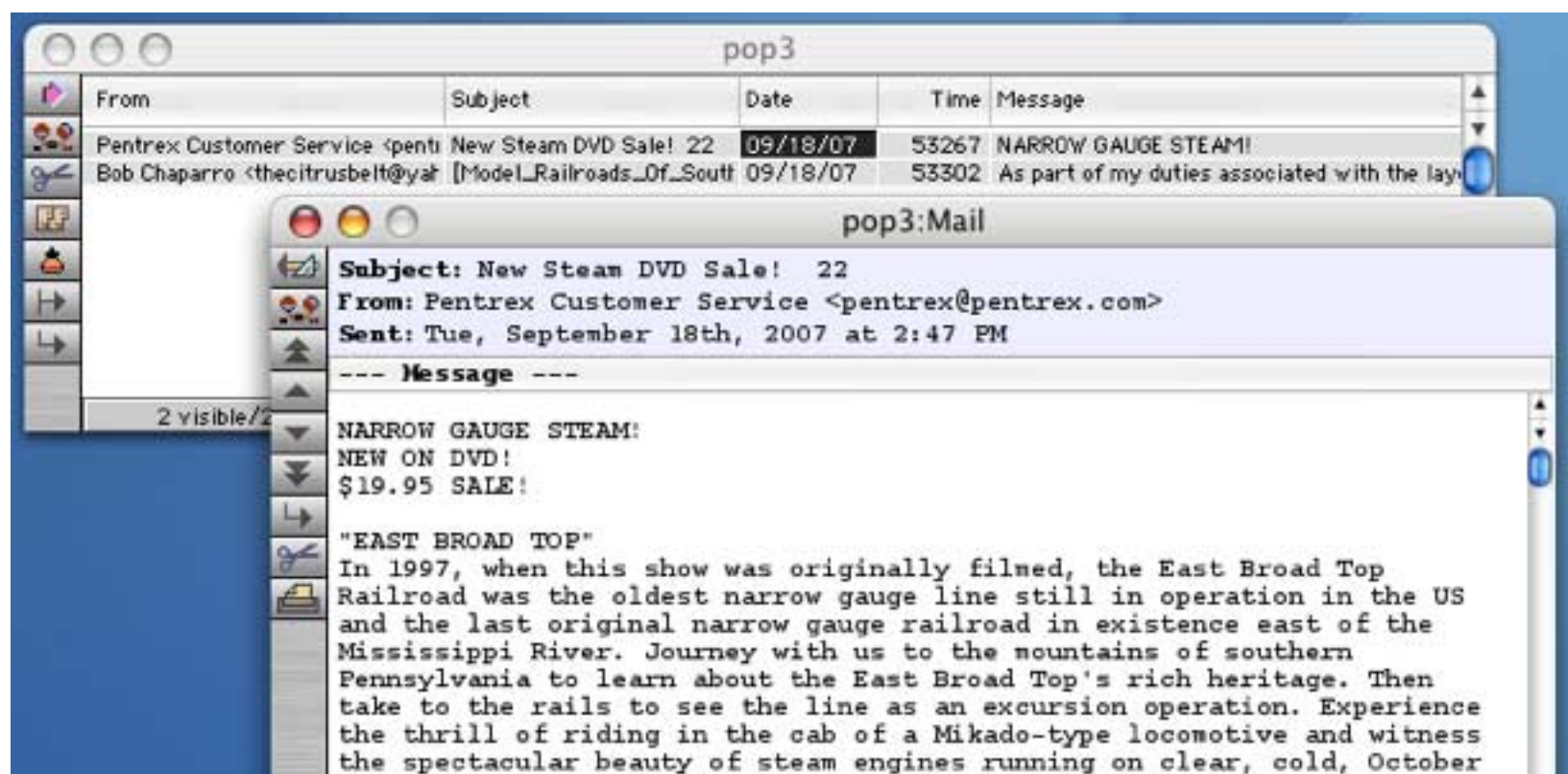
You can also edit the owner or group owner of the file. When you press **Enter** or **Tab** to finish editing this procedure will be triggered:



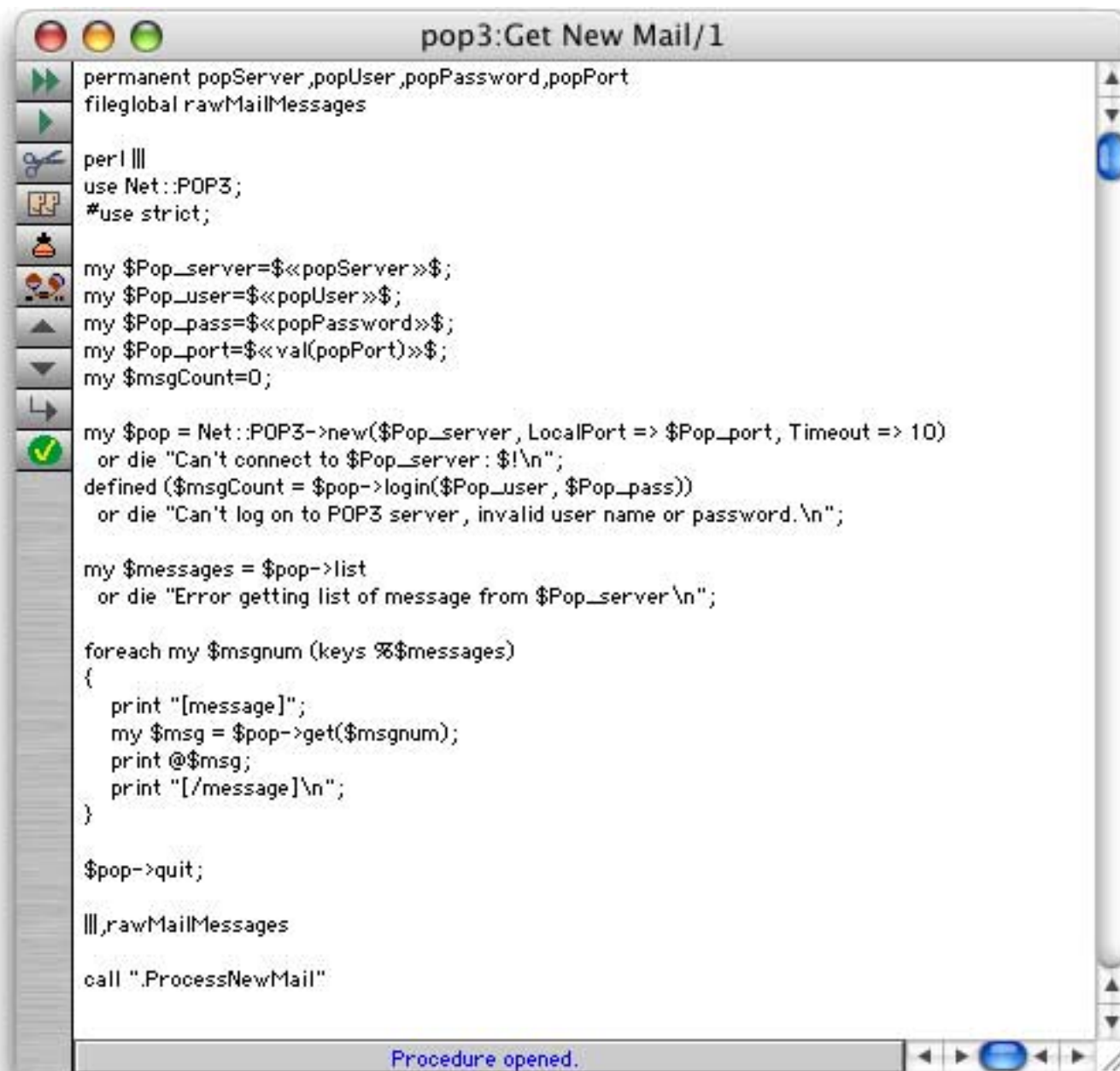
This program uses the unix **chown** command to change the owner. This command can only be run with root privileges, so you must enter the administrator password to run the command. This procedure is set up to remember the password after you enter it once. (In other words, you'll only have to enter the password once each time you open the database, not every time you change the owner of a file.) If you enter the wrong password the **chown** command will fail, which is caught by the **if error** statement. This code is written so that if this happens you'll be prompted to enter the password again.

Perl — POP3 Mail Reader

This example uses embedded Perl code to read mail from a POP3 server.



Here's the program that reads mail from the POP3 server.



```

permanent popServer ,popUser ,popPassword ,popPort
fileglobal rawMailMessages

perl |||
use Net::POP3;
#use strict;

my $Pop_server=$«popServer»$;
my $Pop_user=$«popUser»$;
my $Pop_pass=$«popPassword»$;
my $Pop_port=$«val(popPort)»$;
my $msgCount=0;

my $pop = Net::POP3->new($Pop_server, LocalPort => $Pop_port, Timeout => 10)
  or die "Can't connect to $Pop_server: $!\n";
defined ($msgCount = $pop->login($Pop_user, $Pop_pass))
  or die "Can't log on to POP3 server, invalid user name or password.\n";

my $messages = $pop->list
  or die "Error getting list of message from $Pop_server\n";

foreach my $msgnum (keys %$messages)
{
  print "[message]";
  my $msg = $pop->get($msgnum);
  print @$msg;
  print "[/message]\n";
}

$pop->quit;

|||,rawMailMessages

call ".ProcessNewMail"

```

The program starts by including the `Net::POP3` module, which is part of the standard Perl distribution:

```
use Net::POP3;
```

It then gets the server url, user name, password and server port ID from Panorama variables. In this example these variables have been set up with a separate configuration form:

```
my $Pop_server=$«popServer»$;
my $Pop_user=$«popUser»$;
my $Pop_pass=$«popPassword»$;
my $Pop_port=$«val(popPort)»$;
```

Next the code sets up a connection to the POP3 server:

```
my $pop = Net::POP3->new($Pop_server, LocalPort => $Pop_port, Timeout => 10)
```

and then logs on to the server:

```
defined ($msgCount = $pop->login($Pop_user, $Pop_pass))
```

Assuming the logon is successful it gets a list of the messages waiting to be downloaded (this is a list of message id numbers, not the messages themselves).

```
my $messages = $pop->list
```

Finally it grabs the mail messages themselves.

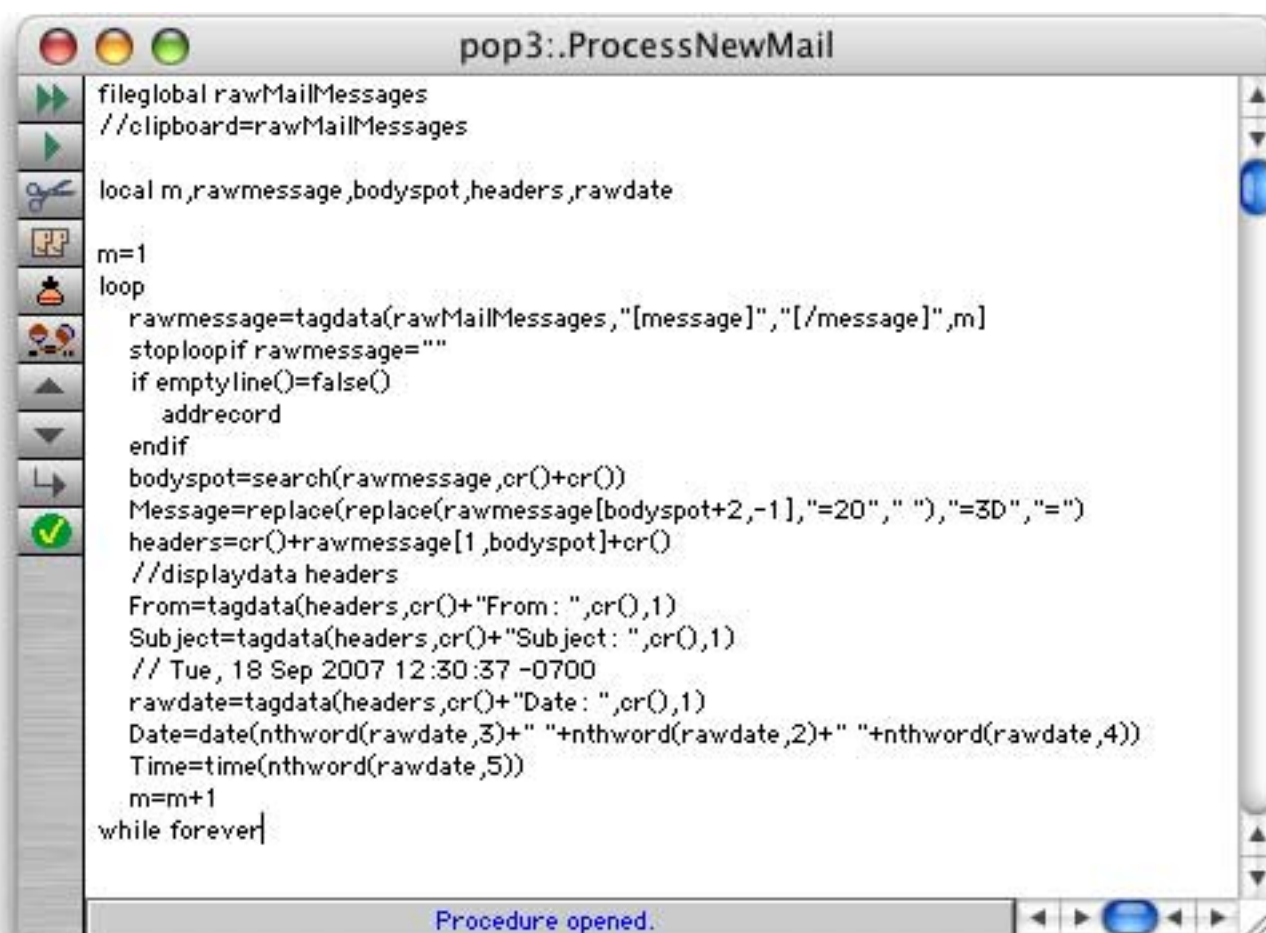
```
foreach my $msgnum (keys %$messages)
{
    print "[message]";
    my $msg = $pop->get($msgnum);
    print @$msg;
    print "[/message]\n";
}
```

The `print "[message]";` and `print "[/message]\n";` statements are included to help Panorama parse the results later. The `print @$msg;` statement actually prints the headers and body of the message (all to standard output).

Finally the program disconnects from the POP3 server:

```
$pop->quit;
```

Once all of this is done a second procedure is called that parses the results that came back from the Perl code.



This procedure uses standard Panorama techniques described in the earlier chapters of this manual.

Ruby — Verify Email Domains

Have you ever had a collection of e-mail addresses and wondered if they were valid? The only surefire check is to actually send mail to each address and see what happens, but you can verify that the domain is valid. Here is a database with domain names in the first column.



Domain	Servers
provue.com	mail.provue.com
apple.com	eg-mail-in2.apple.com, eg-mail-in11.apple.com,
oreilly.com	smtp2.oreilly.com, smtp1.oreilly.com
provoz.com	
tidbits.com	tidbits.com.s5a1.psmt.com, tidbits.com.s5a2.p:

This simple program can check to see if the domain is a valid one for sending e-mail.

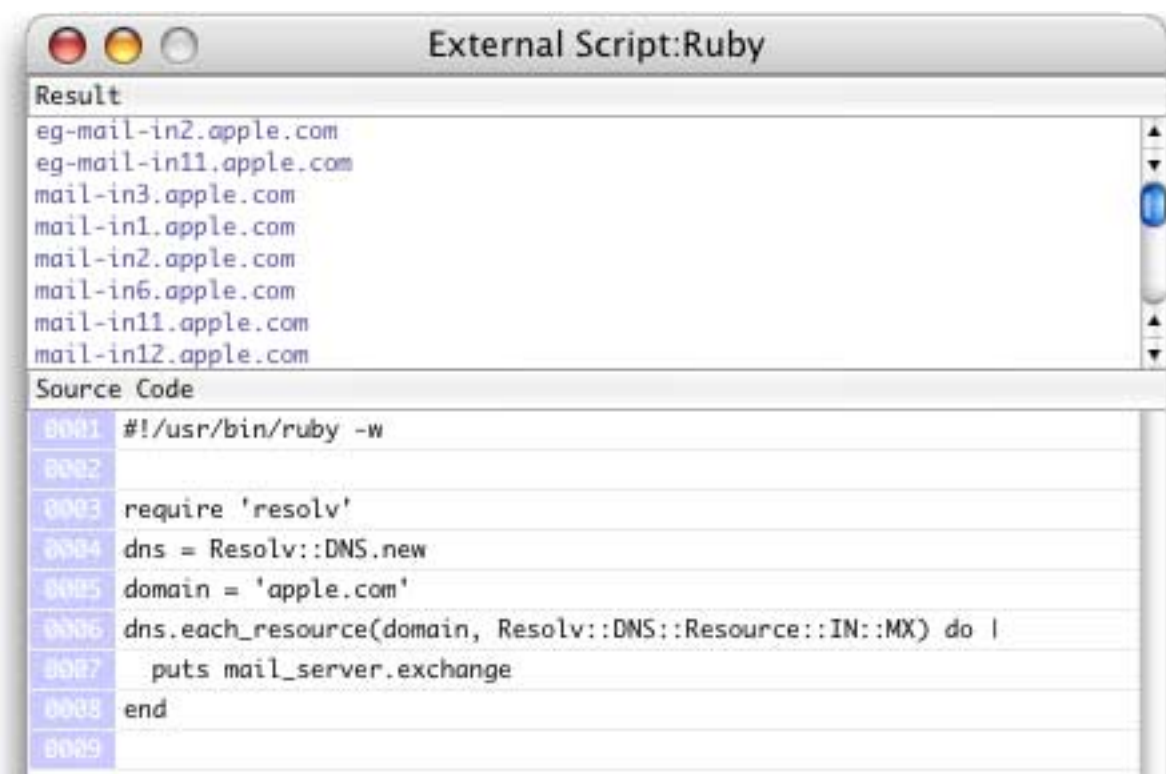


```

ruby |||
require 'resolv'
dns = Resolv::DNS.new
domain = $<<Domain>>$
dns.each_resource(domain, Resolv::DNS::Resource::IN::MX) do |mail_server|
  puts mail_server.exchange
end
|||,Servers
Servers=replace(Servers,or(),"")

```

The Ruby code checks to see if there is a DNS record with MX records for the domain (MX records point to mail servers). The results will look something like this:



```

Result
eg-mail-in2.apple.com
eg-mail-in11.apple.com
mail-in3.apple.com
mail-in1.apple.com
mail-in2.apple.com
mail-in6.apple.com
mail-in11.apple.com
mail-in12.apple.com

Source Code
0001 #!/usr/bin/ruby -w
0002
0003 require 'resolv'
0004 dns = Resolv::DNS.new
0005 domain = 'apple.com'
0006 dns.each_resource(domain, Resolv::DNS::Resource::IN::MX) do |
0007   puts mail_server.exchange
0008 end
0009

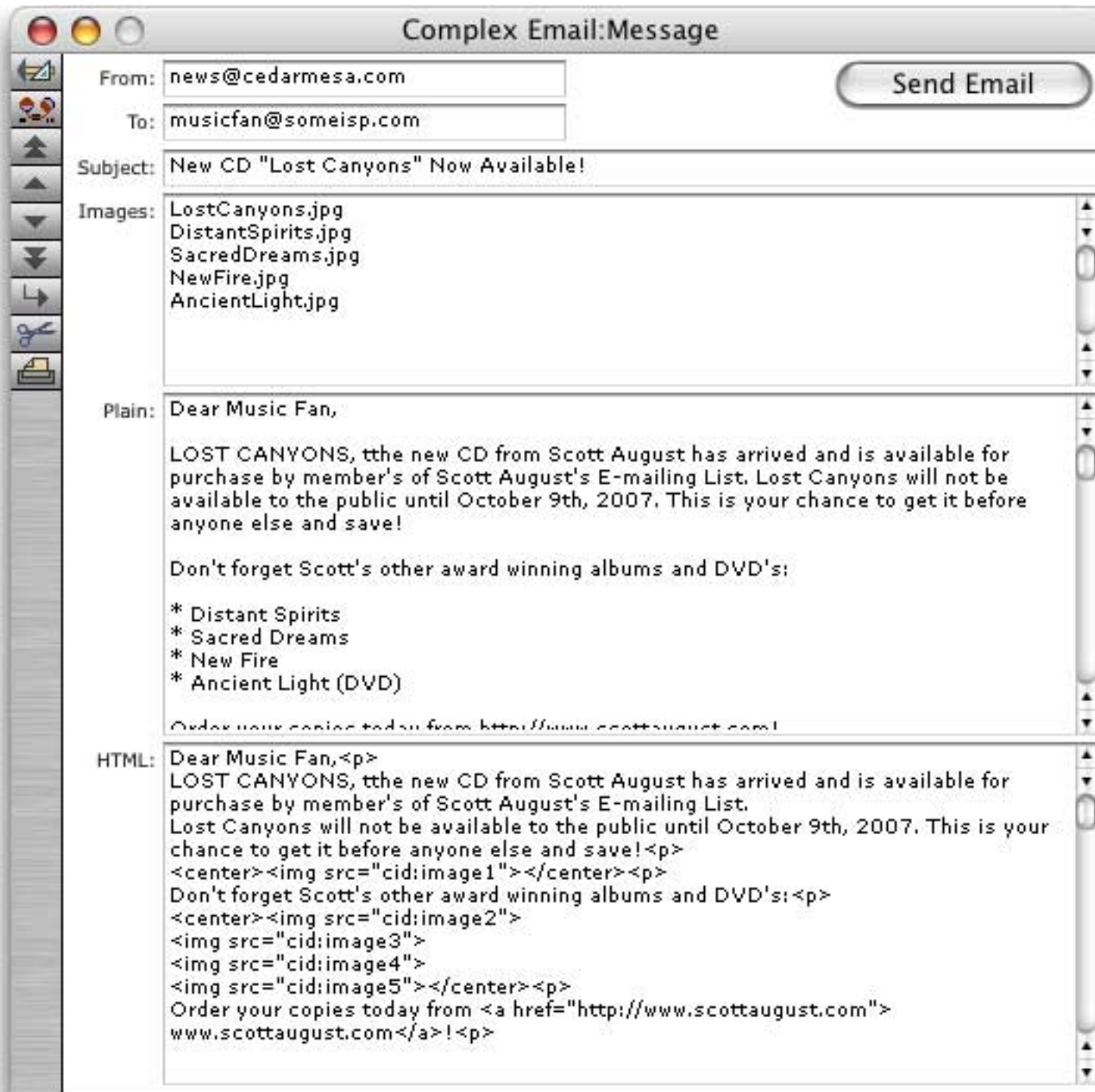
```

Apple has a *lot* of mail servers! If the result is empty then there is no MX record for this domain, so it is not a valid e-mail domain.

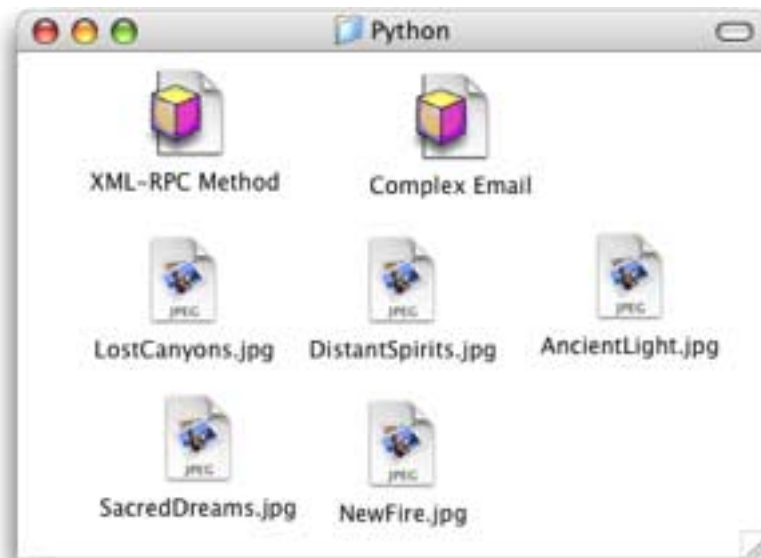
The example database contains a second procedure that checks all of the domains in the database. It basically works the same, but encloses the Ruby code in a Panorama loop to check each record from top to bottom.

Python — HTML + Plain Text Email with Images

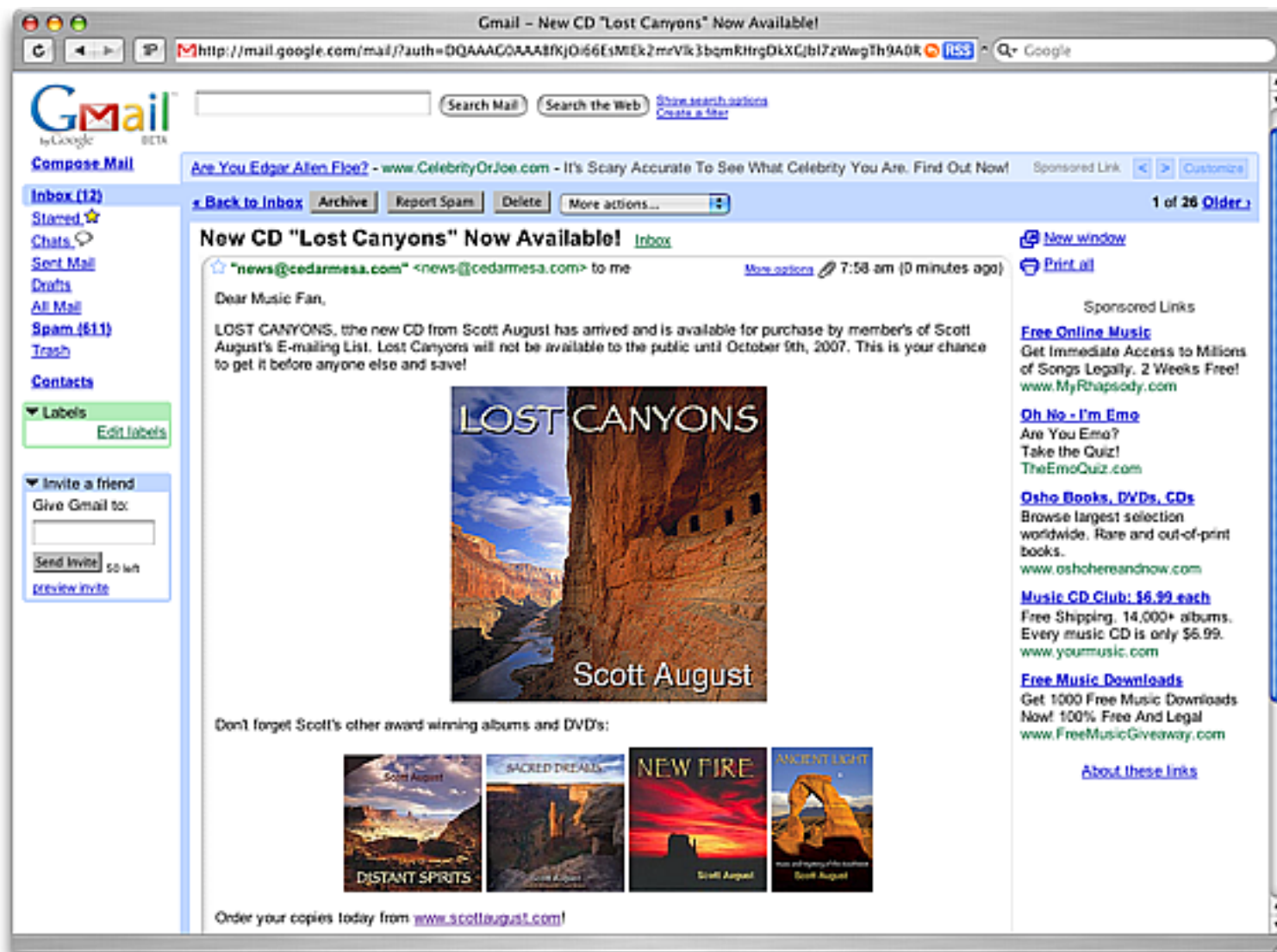
The ability to send plain text e-mail has been a standard feature of Panorama since version 5.0. But what if you want to send HTML e-mail with images? This sample file demonstrates a way to do this using Python. The database fields include a list of images (which must be in the same folder, a plain text version of the e-mail, and an HTML version of the e-mail.



The image files must be stored in the same folder as database.



Let's jump to the end — when you press the **Send Email** button a fancy e-mail is sent, complete with HTML text and pictures.



If you send the e-mail to someone that has a plain text e-mail client they'll see only the plain text.



Here is the complete procedure that actually composes and sends the e-mail.

```

Complex Email:Send Email
/* adapted from http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/473810 */
permanent smtpServer ,smtpUser ,smtpPassword

/* convert to full HFS image paths (assume Images contains paths relative to current db) */
local hfsImagePath ,unixImagePath
arrayfilter Images ,hfsImagePath ,cr() ,dbpath()+import()
/* convert to UNIX path format */
arrayfilter arraystrip(hfsImagePath ,cr() ,unixImagePath ,cr() ,unixshellpath(import()))

python III

from email.MIMEMultipart import MIMEMultipart
from email.MIMEText import MIMEText
from email.MIMEImage import MIMEImage

# Define these once; use them twice!
strFrom = $«From»$
strTo = $«To»$

# Create the root message and fill in the from, to, and subject headers
msgRoot = MIMEMultipart('related')
msgRoot['Subject'] = $«Subject»$
msgRoot['From'] = strFrom
msgRoot['To'] = strTo
msgRoot.preamble = 'This is a multi-part message in MIME format.'

# Encapsulate both the plain and HTML versions of the message body in an
# 'alternative' part, so e-mail clients can decide which they want to display.
msgAlternative = MIMEMultipart('alternative')
msgRoot.attach(msgAlternative)
msgText = MIMEText($«Plain»$)
msgAlternative.attach(msgText)
msgText = MIMEText($«HTML»$, 'html')
msgAlternative.attach(msgText)

# ATTACH MULTIPLE IMAGES
images = $«unixImagePath»$.splitlines()
i=1
for image in images:
    # print 'Image',i,' : ',image,'\n'
    fp = open(image, 'rb')
    msgImage = MIMEImage(fp.read())
    fp.close()
    # Define the image's ID as referenced above
    msgImage.add_header('Content-ID', '<image'+str(i)+'>')
    msgRoot.attach(msgImage)
    i+=1

# Send the email (this example assumes SMTP authentication is required)
import smtplib
smtp = smtplib.SMTP()
smtp.connect($«smtpServer»$)
smtp.login($«smtpUser»$, $«smtpPassword»$)
smtp.sendmail(strFrom, strTo, msgRoot.as_string())
smtp.quit()
III
message "E-mail sent."

Procedure OK.

```

There are three permanent variables that contain the URL of your SMTP server as well as your user name and password on that server. These variables are set up elsewhere, either with a form or in another procedure.

```
permanent smtpServer ,smtpUser ,smtpPassword
```


Next the code prepares the list of images. First it expands each image name into a complete HFS path (for example `LostCanyons.jpg` becomes something like `Disk:Folder:LostCanyons.jpg`). Then the second arrayfilter converts these paths into a format compatible with UNIX (for example `Volumes/Disk/Folder/LostCanyons.jpg`).

```
local hfsImagePaths,unixImagePaths
arrayfilter Images,hfsImagePaths,cr(),dbpath()+import()
arrayfilter arraystrip(hfsImagePaths,cr()),unixImagePaths,cr(),unixshellpath(import())
```

Now the variable `unixImagePaths` contains the list of paths and images in UNIX format. We'll use this list in the Python code momentarily.

On to the Python code, which starts by including the libraries necessary for manipulating MIME multi-part e-mail:

```
from email.MIMEMultipart import MIMEMultipart
from email.MIMEText import MIMEText
from email.MIMEImage import MIMEImage
```

Next it gets the From and To addresses from Panorama:

```
strFrom = $«From»$
strTo = $«To»$
```

Now we're getting somewhere — the next few lines build the e-mail message object and fill in the e-mail header fields.

```
msgRoot = MIMEMultipart('related')
msgRoot['Subject'] = $«Subject»$
msgRoot['From'] = strFrom
msgRoot['To'] = strTo
msgRoot.preamble = 'This is a multi-part message in MIME format.'
```

The message body is added to the object next, in both plain and HTML versions.

```
msgAlternative = MIMEMultipart('alternative')
msgRoot.attach(msgAlternative)
msgText = MIMEText($«Plain»$)
msgAlternative.attach(msgText)
msgText = MIMEText($«HTML»$, 'html')
msgAlternative.attach(msgText)
```

Now for the images. The first step is to get the list of images into a Python array. The Python `splitlines()` function takes that carriage return delimited list from Panorama (the one that we prepared at the top of the procedure) and splits it into separate array elements.

```
images = $«unixImagePaths»$.splitlines()
```

Now a Python loop iterates over each image to add them to the e-mail object. The code reads each image from disk into the `msgImage` object, adds an id (`image1`, `image2`, `image3`, etc.) then attaches the image to the e-mail object.

```
i=1
for image in images:
    fp = open(image, 'rb')
    msgImage = MIMEImage(fp.read())
    fp.close()
    msgImage.add_header('Content-ID', '<image'+str(i)+'>')
    msgRoot.attach(msgImage)
    i+=1
```

Before moving on lets discuss the image id's a bit further. These id's are used within the `` tags in the HTML source to specify what image you want to display where. For example the first image is displayed by including the tag `` in the text, the second with ``, etc. These tags much mach the Python code, so if you changed the Python code to:

```
msgImage.add_header('Content-ID', '<picture'+str(i)+'>')
```

then you would have to use the tags ``, `` etc. in the HTML text.

Ok, back to our e-mail sending program. The e-mail message is complete with all attachments, now all we have to do is send it. Python has a SMTP library that does all of the heavy lifting for us.

```
import smtplib
smtp = smtplib.SMTP()
smtp.connect($«smtpServer»$)
smtp.login($«smtpUser»$, $«smtpPassword»$)
smtp.sendmail(strFrom, strTo, msgRoot.as_string())
smtp.quit()
```

That's it — the e-mail message is on its way!

PHP — Extract EXIF Information from Images

This example is a modified version of the [Image Drops](#) example described in Chapter 16 (see “[Flash Art Image Drag and Drop](#)” on page 779). The [Image Drops](#) example allows images to be added to a database simply by dragging the images from the Finder. This modified version works the same way, but it automatically extracts the **EXIF** information from the image and puts it into the **Notes** field.



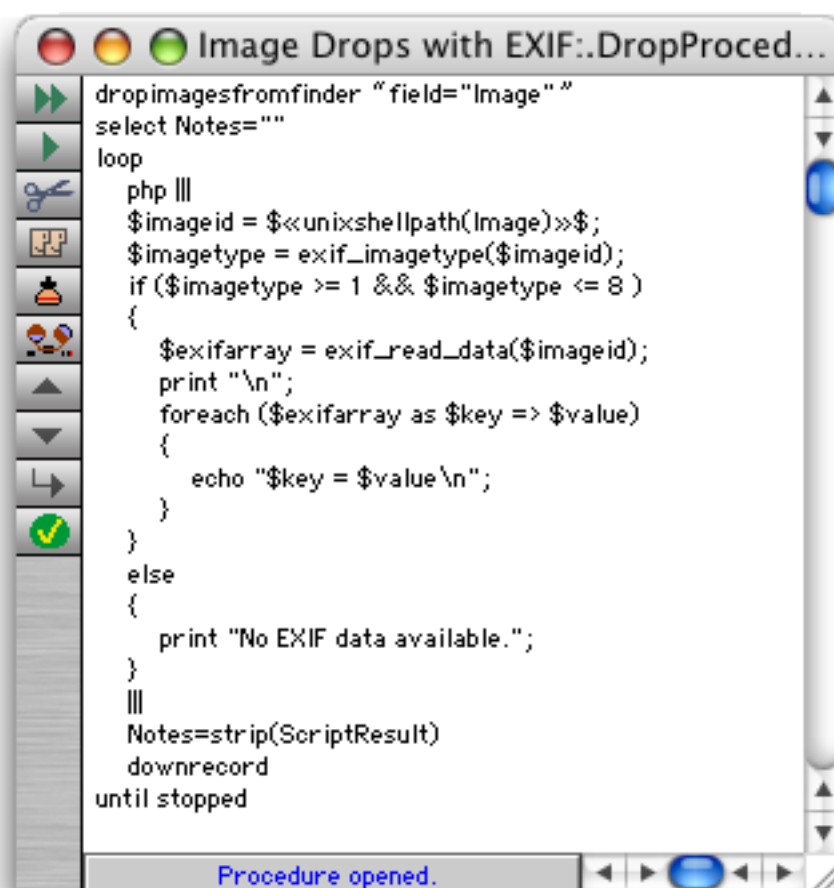
If you are not familiar with **EXIF** this is a message for embedding data within an image. Cameras use **EXIF** to embed detailed information about the image into the image itself. Here is the embedded information for this image, which was taken with an iPhone.

```

FileName = IMG_0043.JPG
FileDateTime = 1185506593
FileSize = 448599
FileType = 2
MimeType = image/jpeg
SectionsFound = ANY_TAG, IFD0, THUMBNAIL, EXIF
COMPUTED = Array
Make = Apple
Model = iPhone
Orientation = 1
ResolutionUnit = 2
DateTime = 2007:07:26 20:23:13
FNumber = 14/5
DateTimeOriginal = 2007:07:26 20:23:13
DateTimeDigitized = 2007:07:26 20:23:13
ColorSpace = 1
ExifImageWidth = 1600
ExifImageLength = 1200

```

Here is the procedure that is triggered when an image file is dropped on the database.



The first line accepts the images and puts the image names and paths into the database. See see “[Flash Art Image Drag and Drop](#)” on page 779 to learn more about the `dropimagesfromfinder` statement.

```
dropimagesfromfinder "field="Image"
```

The rest of the code is a loop that processes all images that have empty an `Notes` field. The procedure starts from the top and works its way through each image. The code uses PHP’s EXIF library, which you can learn more about here:

<http://us.php.net/manual/en/function.exif-imagetype.php>

For each image the first step is to find out what type of image (if any) this is.

```
$imageid = $«unixshellpath(Image)»$;  
$imagetype = exif_imagetype($imageid);
```

The `$imagetype` variable now contains a number denoting the type of image (1=gif, 2=jpeg, 3=png, 4=swf, 5=psd, 6=bmp, 7=tiff, 8=tiff, 9+ = other format). If the type is from 1 to 8 then the EXIF data is extracted from the image:

```
$exifarray = exif_read_data($imageid);
```

To get this data back to Panorama it must be converted from an array into text and printed to standard output.

```
foreach ($exifarray as $key => $value)  
{  
    echo "$key = $value\n";  
}
```

The only remaining step is to copy the data into the `Notes` field after the PHP code is done.

```
Notes=strip(ScriptResult)
```

Voila!

Using AppleScript to Control Panorama from Other Applications

So far we've discussed embedding other languages into Panorama, which is the most common need of most Panorama users. However, it is also possible to embed Panorama code into an AppleScript, allowing Panorama to be controlled by external scripts.

AppleScript is an unusual language in that it is not constant. Instead of creating a complete language, Apple developed only a very skeletal framework. The rest is filled in by the actual application being programmed. The result is that each application is programmed somewhat differently.

If you are an experienced AppleScript programmer, this list describes how Panorama fits into the AppleScript scheme of things. (If this list doesn't mean anything to you, go back and review one of the AppleScript books listed above.)

• Panorama is not recordable.
• Panorama supports the object model for transferring numbers and strings between AppleScript and Panorama.
• Panorama does not support the whose clause.
• AppleScripts can launch Panorama procedures.
• Panorama procedures can be included within a script.
• Panorama procedures can launch AppleScripts.

If you examine this list carefully, you can see that AppleScript really gives you the same control over Panorama that Panorama's built-in programming language gives you. Anything that can be done in a Panorama procedure can also be done within an AppleScript program, and other applications can be programmed as well.

Everything You Really Need to Know...

Although Panorama's AppleScript dictionary includes over a dozen commands that can be used in a multitude of combinations, most scripts involving Panorama boil down to two basic operations: 1) transferring data between AppleScript variables and Panorama fields and variables, and 2) running programs written in Panorama's built-in programming language. The next two pages will show you the easiest methods to accomplish these two operations, and should meet 99.95% of your Panorama AppleScripting needs without even having to read the rest of the chapter.

Value of Cell

Within scripts, you'll use the phrase **value of cell** to access and modify Panorama fields and variables. This phrase must be followed by the name of the field or variable (in quotes). For example, this script checks to see if the **PaymentMethod** field in the current record of the current database is **MasterCard**.

```
tell application "Panorama"
    if Value of Cell "PaymentMethod" = "MasterCard" then
        -- process master card
    end if
end tell
```

The **value of cell** phrase may be used to access either fields or global variables. When it is used to access a field, that field must be in the currently active database, i.e. the database with the frontmost window within Panorama. (However, Panorama itself does not have to be the active application.)

Using the AppleScript **set** statement, a script can copy Panorama database fields (or variables) into AppleScript variables. This example pulls the name and address out of the current database into the AppleScript variable **LabelText**, then makes a label in WordPerfect.

```
tell application "Panorama"
    set LabelText to -
        Value of Cell "Name" & return & -
        Value of Cell "Address" & return & -
        Value of Cell "City" & ", " & -
        Value of Cell "State" & space & -
        Value of Cell "Zip"
end tell
tell application "WordPerfect"
    copy LabelText
        to beginning of paragraph 1
end tell
```

By reversing the order of the parameters, the **set** statement can be used to copy data into Panorama fields or variables. This example gets the name of the topmost window in the Finder, then puts that name into a field named **Folder** in the current database. (If there is a global variable named **Folder**, the name will go into the variable instead of into a field.)

```
tell application "Finder"
    set ActiveFolder to name of window 1
end tell
tell application "Panorama"
    set Value of Cell "Folder" to ActiveFolder
end tell
```

It is also possible to access database cells by number instead of by name, although this is rarely of any use. For example, to get the value of the first field in the database use value of cell 1, for the second field value of cell 2, etc.

Executing Panorama Procedures

Using the **execute** statement, you can put a Panorama procedure right in the middle of any AppleScript. Simply type the procedure in quotes after the word **execute**. This example tells Panorama to open the form **Shipping**.

```
tell application "Panorama"
    execute "openform "Shipping""
end tell
```

This example includes a single Panorama statement, but your procedure may be as complex as you wish. Notice, however, that since the entire procedure must be surrounded by double quotes, you cannot use double quotes within your procedure. There are several solutions to this: 1) you can use smart quotes, as shown above, 2) you can use curly braces {} instead of quotes, 3) you can use single quotes instead of double quotes, or 4) you can use \ for each double quote. All four of these methods are shown in this example:

```
tell application "Panorama"
    execute "openform "Shipping""
    execute "openform {Shipping}"
    execute "openform 'Shipping'"
    execute "openform \"Shipping\""
end tell
```

The **execute** statement is not limited to a single statement or a single line. It can include complex procedures like this:

```
tell application "Panorama"
  Execute "field {Machine Type}
  formulafill array(SystemInfo,1,{-})
  field {System Version}
  formulafill array(SystemInfo,2,{-})
  groupup
  field {Machine Type}
  count
  outlinelevel 1"
end tell
```

The **execute** statement does not make Panorama the frontmost application. If the procedure is going to display a dialog or allow the user to interact with a window, the script should activate Panorama (bring it to the front) before using the execute statement. To bring Panorama to the front, use the AppleScript **activate** statement.

Transferring Data Between AppleScript and a Panorama Program

Using the **value of cell** phrase, it's easy to transfer data between AppleScript and the procedure in the **execute** statement. This example uses Panorama's **dbinfo()** function to get a list of the fields in the currently active database.

```
tell application "Panorama"
  execute "global zFieldList
  zFieldList=dbinfo("fields","")"
end tell
set dataFields to value of cell "zFieldList"
if dataFields contains "Address"
  (* process address ... *)
end if
```

Here is a script that passes data to a Panorama procedure. This script is designed to be saved as an application. When you drag and drop a text file (or files) on this application it will automatically import and append the text files into the current database.

```
on open fileList
  tell application "Finder"
    repeat with oneFile in fileList
      set filePath to oneFile as string
      tell application "Panorama"
        Execute "global importFile"
        set Value of Cell "importFile" to filePath
        Execute "openfile {+}+importFile"
      end tell
    end repeat
  end tell
end open
```

Note: This example requires the Scriptable Finder, which is included with System 7.5 or later.

Transferring a Value Back From Panorama to the AppleScript (Returning a Value)

The procedure triggered by execute can return a value back to the AppleScript that called it. This is done with the **SetAppleEventValue** statement.

```
setappleeventvalue value
```


The **value** parameter is the value to return to the calling AppleScript. This value must be either text or an integer (dates, floating point and non-integer fixed point values are not allowed). These examples return a constant value, but any formula may be used.

```
setappleeventvalue "alpha"
```

```
setappleeventvalue 3
```

If the **setappleeventvalue** is used more than once within a procedure the last value set will be returned.

This example counts the number of open files in Panorama and sets the AppleScript variable **dbcount** to that value.

```
tell application "Panorama"
    set dbcount to execute "setappleeventvalue linecount(info({files}))"
end tell
```

Here is a very similar example that sets the AppleScript variable **dblist** to a list of currently open files.

```
tell application "Panorama"
    set dblist to execute "setappleeventvalue info({files})"
end tell
set AppleScript's text item delimiters to return
set dblist to every text item of dblist
```

See the next section for an explanation of the last two lines of this example.

Working with Lists

One of AppleScript's powerful features is the **List** data type. Panorama does not directly support the list data type, but you can easily convert between Panorama text arrays and lists, and back again.

Here is an example that transfers a text array to an AppleScript variable and then converts that variable into a list.

```
tell application "Panorama"
    Execute "global aString
    aString=dbinfo({fields},{})"
    set databaseFields to Value of Cell "aString"
end tell
set AppleScript's text item delimiters to return
set databaseFields to every text item of databaseFields
```

This example gets a list of all the currently running programs (called processes) and then converts that list to a comma separated text array.

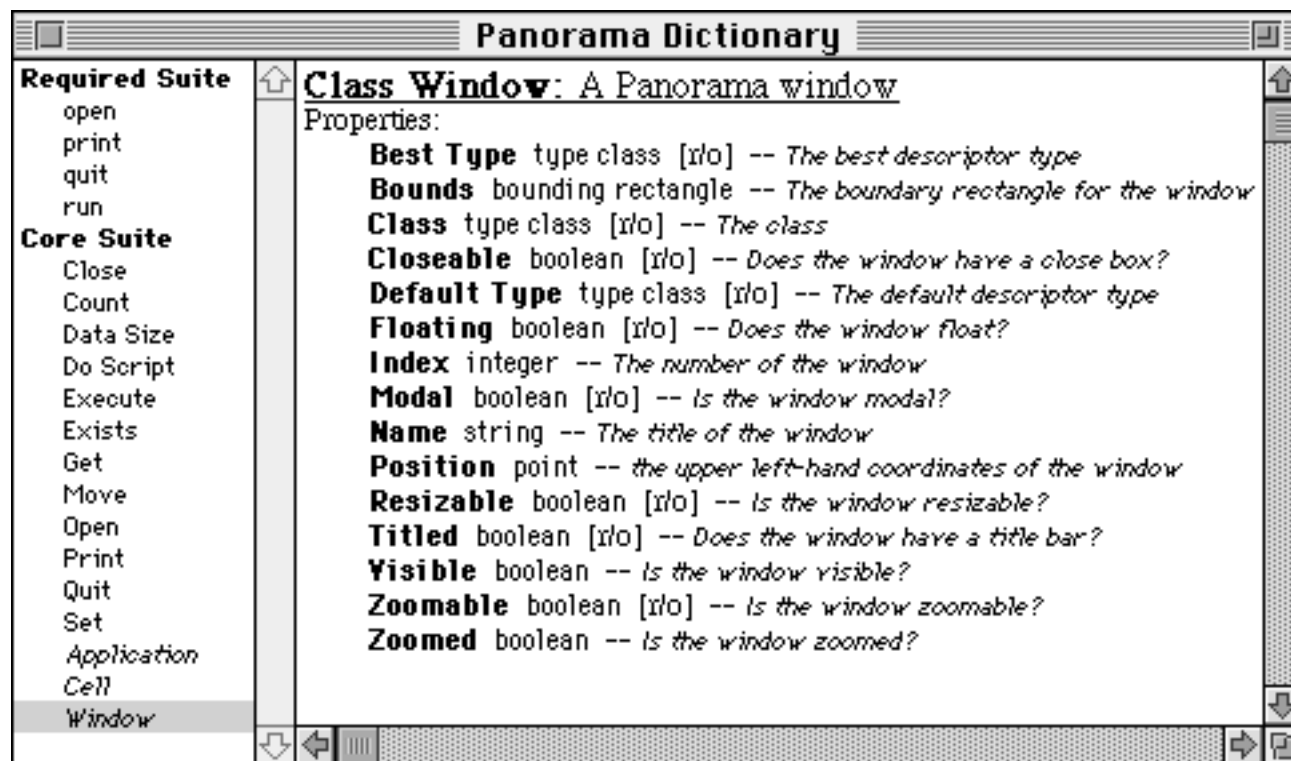
```
tell application "Finder"
    set ProcessList to name of every process
end tell
set AppleScript's text item delimiters to ","
set ProcessList to ProcessList as text
```

The script could continue by passing the text array to Panorama for further processing. (Note: This example requires the Scriptable Finder, which is included with System 7.5 or later.)

AppleScript & Panorama... The Rest of the Story

The previous pages cover everything you really need to know to do work with AppleScript and Panorama. However, there is more to the story. Most of the material that follows really falls into the category of “more ways to do the same things” and is not really vital.

Our guide throughout the rest of this appendix is the Panorama AppleScript dictionary. You can open and view this dictionary from within the AppleScript **Script Editor** application.



The Required Suite

Like all other AppleScript savvy applications, Panorama supports the four required statements: **open**, **close**, **quit**, and **run**.

The **open** statement opens one or more database files. This statement requires one parameter, a reference to one or more files. If you want to reference a single file, simply include the word file followed by the name of the file in quotes.

```
tell application "Panorama"
    Open file "Address List"
end tell
```

Notice that in this example, the word **file** is part of the parameter, not part of the command. This is different from Panorama’s **openfile** statement, which is all one word. (Of course you could also use the **openfile** statement to open files, as shown here:)

```
tell application "Panorama"
    execute "Openfile {Address List}"
end tell
```

The **print** statement prints one or more database files. This statement is identical to selecting the files in the Finder and choosing **Print** from the File menu. However, this method of printing give you no control over what form is used for printing, or what records are selected for printing. To get control over these parameters, use the **execute** statement with Panorama's **print** statement.

```
tell application "Panorama"
    activate
    execute "Openfile {Address List}
    openform {My Report}
    select Date>date({1/1/96})
    print dialog"
end tell
```

The **quit** statement shuts down Panorama. The **run** statement starts Panorama up if it is not running. However, this statement is not really very useful because Panorama will start up automatically any time a script asks it to do something by including the phrase `tell application "Panorama"` in the script.

The Core Suite

The Core Suite includes 9 additional statements you can use in your scripts.

The **close** statement closes a window. Unlike Panorama's **close** statement, the AppleScript **close** statement can close any window, not just the top window. For example, to close the 3rd window from the top, use this script:

```
tell application "Panorama"
    close window 3
end tell
```

The **count** statement counts windows or fields. Here is a simple script that counts the current number of open windows in Panorama.

```
tell application "Panorama"
    get Count windows
end tell
```

A slightly different format is required to count fields. This script counts the number of fields in the currently active database.

```
tell application "Panorama"
    get Count of every Cell
end tell
```

(Note: This **of every** format works with windows also, one of the many examples of redundancy in the AppleScript language. This redundancy is not consistent, however, and you cannot say `get Count cells` to find the number of fields in a database. Of course you could also create these examples with the **execute** statement, using Panorama itself to count the windows or fields.)

The **data size** statement can be used to get the size of the contents of a field or variable. For example, suppose the field **Name** in the current record contains John. In that case, the result of this script will be the number 4 (the number of characters in the name John).

```
tell application "Panorama"
    get Data Size of Value of Cell "Name"
end tell
```

The **do script** statement launches a Panorama procedure. The procedure must be pre-defined in the current database.

```
tell application "Panorama"
    do script "Year End Totals"
end tell
```

The **execute** statement lets you put a Panorama procedure inside an AppleScript. Unlike the **do script** statement, the **execute** statement requires no advance preparation in the database itself.

The **exists** statement can be used to determine if a field, variable, or window exists or not. This statement returns a true-false result, and is usually used with the **if** statement. This script checks to see if the current database has a field called **Name**. If it does contain such a field, the script converts the field to all upper case.

```
tell application "Panorama"
  if Exists of Cell "Name" then
    execute "field Name
    formulafill upper(Name)"
  end if
end tell
```

The **get** statement is the standard AppleScript get statement—it simply evaluates a value and puts it in the **Result** variable. In the script editor you can open the **Result** window to see the contents of this variable, which can be useful for debugging.

The **move** statement works with windows. Using this statement, you can change the order of the windows within Panorama. Here are some examples of how the move statement can be used.

```
Move Window 2 to beginning

Move Window 1 to end

Move Window 1 to after Window 2

Move Window 1 to before Window 5

Move Window 1 to back of Window 2
```

The **set** statement is the AppleScript equivalent of the assignment statement in most programming languages. For example, in most programming languages you would add two numbers like this:

```
Sum=3+4
```

But in AppleScript this assignment is written like this:

```
set Sum to 3+4
```

The **set** statement is one of the most frequently used statements in the AppleScript language.

The Objects

Panorama has three types of objects that you can use in your scripts: **application**, **window**, and **cell**.

The **application** object refers to Panorama itself. You cannot modify this object, but you can get useful information about it—the version, name, etc. This script prints the current database, but only if Panorama is the topmost application.

```
tell application "Panorama"
  if frontmost
    execute "print dialog"
  end if
end tell
```

(Note: One of the properties of an application is its version number. The dictionary says that this is a string. However it is not a string or a number, but a special class. There is not much you can do with this special version class. Unfortunately, this is consistent with other applications, including the Scriptable Finder.)

The window object refers to Panorama windows. A window may be identified by its name or by a number (with 1 being the topmost window). Here is a script that gets the name of the topmost window:

```
tell application "Panorama"
    get Name of Window 1
end tell
```

In addition to the name, there are many other properties of a window that you can access: the window location, its size, whether or not it has a close box or zoom box, and many more. See the Panorama AppleScript dictionary for a complete listing of window properties.

Some window properties can be changed from AppleScript with the **set** statement. For example, you can move a window to a new position using the **bounds** property.

```
tell application "Panorama"
    set Bounds of Window 1 to {50, 100, 400, 250}
end tell
```

You can change the order of windows with the **move** statement. This script moves the third window to the front.

```
tell application "Panorama"
    activate
    Move Window 3 to beginning
end tell
```

The **activate** statement is not actually necessary. It brings Panorama to the front, which makes it easier to see the windows change order.

The **move** statement is very flexible for changing the order of windows. Here are some more examples of possible options.

```
Move Window 2 to beginning

Move Window 1 to end

Move Window 1 to after Window 2

Move Window 1 to before Window 2

Move Window 1 to back of Window 2
```

The **cell** object type is used for working with Panorama fields and variables. The most commonly used property of cell objects is their value, as seen throughout this appendix (Value of cell "Name", etc.)

Another cell property is the index, or field number. (This property only applies to fields, not variables.) Here is an example that gets the field number (1, 2, 3, etc.) of the field **City**.

```
tell application "Panorama"
    get Index of Cell "City"
end tell
```

If the database contains fields called **Name**, **Address**, **City**, **State** and **Zip** then this script will return the value **3**.

Another cell property is the cell **Name**. Again, this really only applies to fields, not variables. This script uses the **Name** property to build a list of all the fields in a database.

```
tell application "Panorama"
  set CellNames to {}
  repeat with cellnumber from 1 to Count of every Cell
    set CellNames to (CellNames & Name of Cell cellnumber)
  end repeat
  get CellNames
end tell
```

The script above will work fine and illustrates the **Name** property well, but by letting Panorama itself do some of the work we can create a script that runs much faster.

```
tell application "Panorama"
  Execute "global aString aString=dbinfo("fields","")"
  set databaseFields to Value of Cell "aString"
end tell
set AppleScript's text item delimiters to return
set databaseFields to every text item of databaseFields
get databaseFields
```

The dictionary lists several other properties of cell objects (Best Type, Class, Default Type) but these really aren't of any use to AppleScript programmers (although they are used internally by AppleScript itself).

Chapter 5: Cross Platform Databases



Most Panorama databases can be prepared for cross platform operation in a few seconds. In fact, for most files the process of transferring a file from the Macintosh to the PC is as simple as adding **.pan** to the file name and transferring the file to the PC.

File Type/Creator vs. Extensions

The Mac OS uses an invisible 8 character designator to identify the type of each file. The designator is divided into a 4 character **type code** and a 4 character **creator code**. If you are not a programmer you may not have ever realized these codes were there, because they are completely hidden.

The Windows operating system does not have an invisible designator to keep track of file types. Instead, Windows uses a visible designator appended to the end of the file name. This designator, called an **extension**, is a period followed by 3 or 4 characters. For example, **.txt** is a text file, **.exe** is a program file (application), and **.pan** is a Panorama database. If a filename doesn't have an appropriate extension, Windows can't tell what kind of file it is.

Although there are hundreds of different extensions, there are only a handful that apply to Panorama databases and their associated files.

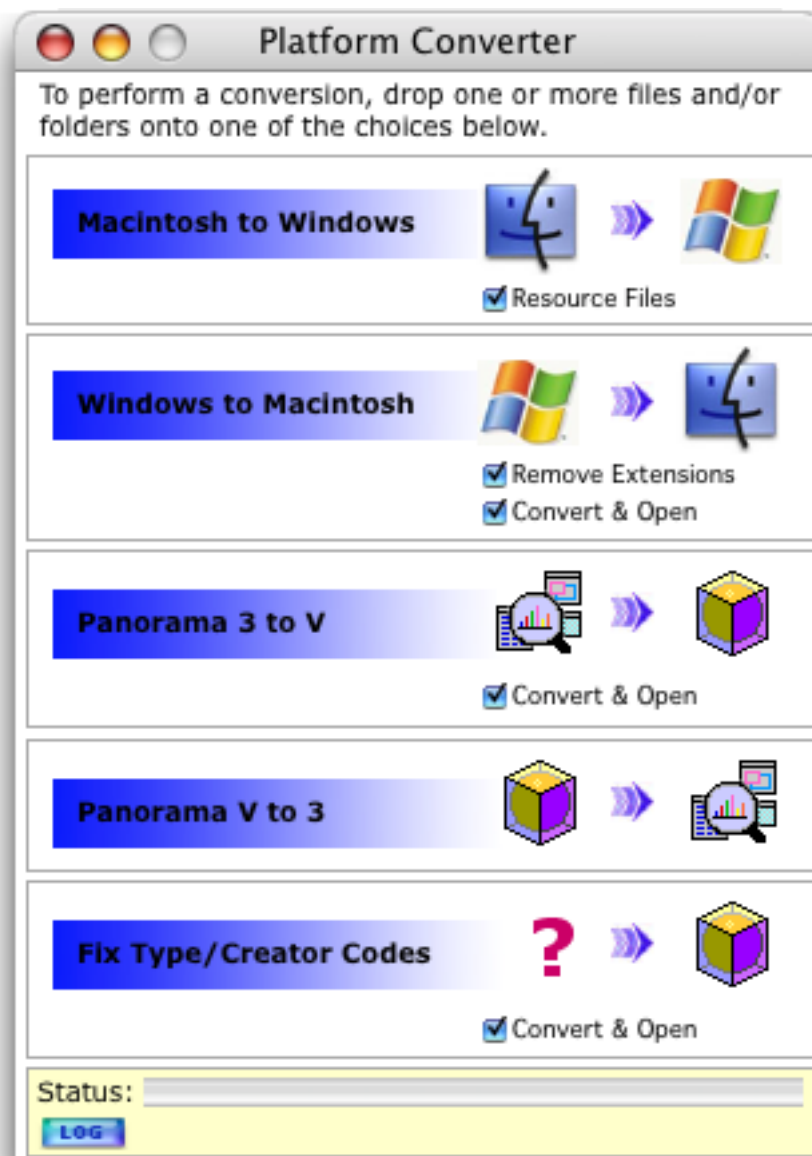
Extension	Type of File
.pan	Panorama Database
.pnz	Panorama File Set
.pwp	Panorama Word Processor File
.pct	Macintosh Picture (PICT) File
.rsr	Macintosh style Resource File
.txt	Text File

Before a file may be used on the PC, it must have the correct extension added. If you only have a few files, you may wish to simply type in the extensions yourself.

Since the Macintosh version of Panorama does not normally use extensions, we've mostly tried to hide them within Panorama itself. For example, if you open a database named **Checkbook.pan**, Panorama will display it simply as **Checkbook** in the window title, without the **.pan** extension. When you save a file with the **Save As** dialog, it is not necessary to type **.pan** if the file original had a **.pan** extension -- Panorama will add the extension for you.

Platform Converter Wizard

If you have more than a few files to convert you can use the **Platform Converter** wizard to help. The converter can automatically convert an entire folder of files (including subfolders, if any). The converter examines the hidden type and creator codes for each file, and automatically adds the appropriate extension. The Platform Converter can also convert Macintosh resource files for you (more on that later). If you are using MacOS you'll find the **Platform Converter** wizard in the **Utilities** submenu of the **Wizard** menu. (The **Platform Converter** wizard is not available on Windows PC systems because by the time a file has been copied to the PC, the hidden codes needed by the converter are gone. Platform conversion must be performed on a Macintosh system, since that's where the hidden codes are accessible.)



Converting from Macintosh to PC

To convert a file, group of files, or folder full of files from Mac to PC simply select the files in the Finder, then drag them and drop them on the **Macintosh to Windows** section of the wizard.



The converter will scan each file and folder that was dropped (and subfolders, if any). Based on the hidden file type and translation, the converter will add the appropriate extension to each file. As it performs the conversion, the program keeps a log of everything it does. You can see the log by pressing the **Log** button. The log also shows any errors encountered by the converter. Typical errors include a file name that would be more than 31 characters long with the extension added (the PC allows 255 but the Mac only allows 31), a file name that contains characters not allowed by the PC (for example slash or backslash), or a file type that cannot be converted to the PC (an application, for instance).

Converting Resources

A Macintosh file may actually contain two separate components, called the data fork and the resource fork. The resource fork can be used to hold custom menus, icons, text, pictures, and other items. Windows files, however, only contain one "fork," which corresponds to the data fork.

The Panorama Platform Converter, however, can convert any file that contains a resource fork so that it may be used on the PC. If the **Resource Files** option is checked, the converter will check each file to see if it contains a resource fork. If it does, the converter will copy the resource information into the data fork of a new file, with an extension of **.rsr**. The PC version of Panorama knows how to access the items (custom menus, etc.) that are stored inside the **.rsr** file.

Converting from PC to Macintosh

If you create a Panorama database on the PC and then copy it back to the Mac, the new file will appear as a generic icon on the desktop. To convert a file, group of files, or folder full of files from PC to Macintosh simply select the files in the Finder, then drag them and drop them on the **Windows to Macintosh** section of the wizard.



The converter will scan each file and folder that was dropped (and subfolders, if any). Based on the extension (.pan, .pnz, etc.) it sets the proper hidden type and creator code for each file, allowing the file to be accessed properly on the Macintosh. If the **Remove Extensions** option is checked the converter also removes the extension from the file name. It does not, however, convert **.rsr** files back into Macintosh resource files. If the **Convert & Open** option is checked then Panorama will open any databases after they are converted. (Note: Databases converted from PC to Mac can only be opened with Panorama 4.0 or later. Older versions of Panorama will not open these files!)

Converting from Panorama 3.x to V (Macintosh)

Panorama 3.1 and Panorama V can both co-exist on the same Power Macintosh computer, and in fact both can be running at the same time! Databases initially created with Panorama 3.1 will automatically open Panorama 3.1 when double clicked; databases created with Panorama V will automatically open Panorama V when double clicked. Panorama V can open databases created with Panorama 3.1 or earlier simply by dragging these files onto Panorama V or by using the **Open File** dialog. If you want Panorama V to launch automatically when a Panorama 3.1 database is double clicked, you must convert it using the **Platform Converter** wizard. After the database is converted it's icon will change from the old Panorama icon to a new icon.

To convert databases files so that they will automatically launch Panorama V instead of 3.1 select the files in the Finder, then drag them and drop them on the **Panorama 3 to V** section of the wizard.



The wizard will scan the files and change each one to launch Panorama V instead of Panorama 3. If the **Convert & Open** option is checked then Panorama will immediately open the databases after they are converted.

Technical Note: The Panorama 3 to V conversion does not actually change the internal structure of the file. Panorama 3, 4 and V share the exact same internal file structure. The only modification this conversion makes is to change the hidden Type and Creator codes that the Macintosh OS uses to identify the application to be launched when a file is double clicked.

Converting from Panorama V to 3.x (Macintosh)

To convert databases files so that they will automatically launch Panorama 3 instead of V select the files in the Finder, then drag them and drop them on the **Panorama V to 3** section of the wizard.



The wizard will scan the files and change each one to launch Panorama 3 instead of Panorama V. Note: If a database was last opened on a Windows computer Panorama 3.1 will not be able to open the file, and it will not be converted.

Fixing a Database With Missing or Incorrect Type/Creator Information

The Macintosh operating system normally maintains special invisible codes that identify the type of each file, and the application that created that file. These are called type and creator codes. Sometimes, however, these invisible codes can get stripped from a file. When this happens the file will have a generic or incorrect icon. One of the most common ways for this to happen is when sending a file via e-mail — some e-mail programs do not correctly transmit the type and creator codes. It can also happen if the file is copied onto a PC formatted disk, then back to the Macintosh.

If a Panorama database loses its type and creator code, you can restore these codes by dragging the file (or files) onto the **Fix Type/Creator Codes** section of the **Platform Converter** wizard.



Any file that is dropped on this section will have its type and creator codes modified to the correct codes for a Panorama database. **Be sure that you only drop files that you are sure are Panorama databases on this section!** If you drop a file that is not a Panorama database on this section, it will be modified so that its type and creator code will fool the system into thinking that it is a Panorama database. Double clicking such a file will cause Panorama to try to open the file thinking that it is a database, and may cause Panorama to crash.

Sharing Databases Across a Cross Platform Network

In addition to transferring files back and forth between a Mac and a PC, you can actually share a database (or collection of databases) across a cross platform network. It's annoying on the Macintosh, but if you want to use a database on both the Mac and PC, you must include the extension as part of the filename (.pan, etc.) even when you are using the file on the Macintosh. To help keep the transition smooth, the Macintosh version of Panorama slightly modifies its behavior when it detects a database with the .pan extension. Just as when using the PC version of Panorama, the extension is removed for internal use. So if you open a file named **Checkbook.pan**, the window title will simply be **Checkbook**, without the .pan extension. When a file that was originally opened with the .pan extension is saved, the .pan extension is automatically added to the filename, whether you use **Save**, **Save As**, or **Save A Copy As**. The main goal is to keep any existing procedures that reference file names working without changes whether there is an extension or not.

Cross Platform vs. Older Versions of Panorama

The processor used in Windows computers (x86/Pentium) stores numbers in a different format from the processors used in Macintosh computers (PowerPC/68K). Since Panorama files contain many numbers, a conversion must be performed when a database is moved to a different platform. Let's suppose a database is created on a Macintosh. The first time the database is opened on any Windows machine, the numbers inside the database are automatically converted to PC format in memory. The conversion only takes a split second, so Panorama doesn't even notify you that the conversion is happening. When the file is saved, the file with the converted numbers is written to disk. When you re-open the file on the PC, no further conversion is necessary. However, if you transfer the file back to a Macintosh computer and open it, Panorama must re-convert the numbers in the file. Again, this happens automatically, and in a split second. In fact, the whole process is so transparent, you'll never notice it with one exception. The exception is if you attempt to open a file that has been saved on the PC on an older (Panorama 3) version of Panorama. Since older versions of Panorama do not have the conversion code, they will be unable to open the file. For now, the only solution is to make sure you open and save the file on a Macintosh computer before attempting to use the file with an older version of Panorama.

Cross Platform Font Usage

If a font has the same name on the Macintosh and the PC then it can be used in a database on either type of computer. If the database is transferred from a Macintosh to the PC or PC to Macintosh the font will continue to work properly.

Panorama has special handling for four special fonts.

Macintosh	Windows
Geneva	Alpine
New York	Yankee
Chicago	City
Monaco	Block

On the Macintosh these four fonts are always present as universal fonts, so you can rely on them always being available. We have created the four equivalent fonts for Windows computers to guarantee that these fonts are always available on any computer. For example, if you create an object using the Geneva font on a Macintosh computer it will automatically be translated to the Alpine font when displayed on a Windows PC computer. If you want to make sure that your database will display properly on any computer you should restrict yourself to using only these four fonts.

Cross Platform Programming

If you've been doing Panorama programming with a previous version of Panorama, you're probably wondering what it will take to get your procedures and formulas to work cross platform. The good news is, probably nothing! We've gone to great lengths to make Panorama for the PC completely compatible with previous versions, as you'll see below. The payoff is that we have successfully ported several large Panorama applications to the PC without making a single change to the applications. No procedures were changed, no forms, fields, nothing. These applications include the Panorama 3 MegaDemo, Power Team, and several complex third party applications, including one with dozens of files and hundreds of forms and procedures.

So far we have encountered only one database that required changes to work on the PC - the Panorama On-Line Reference. The changes required about 10 minutes to complete, and were needed because several procedures referred to subfolders named •Statements, •Functions(, etc. Unfortunately, the • character is not allowed in a Windows file or folder name, so it had to be changed. Basically, unless your database uses special Macintosh only features (the System folder, AppleScripts, special Apple only characters) you shouldn't have to touch your databases at all, just add extensions and go!

File Name Extensions and the OpenFile Statement

Windows files have extensions (.txt, .pan, etc.) and Macintosh files do not. We've programmed Panorama for the PC so that in almost all cases, your existing procedures will work just fine even if they open other databases. For example, suppose you have a database named [Checkbook](#), and you want to open it inside a procedure. It's simple, right? Just use the `openfile` statement:

```
openfile "Checkbook"
```

However, on the PC, the file that is opened is actually named [Checkbook.pan](#). Don't worry, however -- Panorama will automatically add the extension for you. You don't have to change your procedure at all, and it will automatically work on either the Macintosh or the PC. By the way, it's ok to include the extension if you wish:

```
openfile "Checkbook.pan"
```

However, this code is not portable. It will not work on the Macintosh unless the file is actually named [Checkbook.pan](#).

By the way, there is one case where the .pan extension will be automatically added even if you are on the Macintosh. If the currently open file has a .pan extension, Panorama will assume that the file you want to open has a .pan extension. This allows you to build a set of files that can be shared cross platform on a server (which must all have a .pan extension, even when used on the Mac).

Confused? Don't be. The bottom line is you should pretty much always be able to leave off the extension when using the `openfile` statement.

When programming on the Macintosh you can use the `nodefaultextension` statement to open a database that doesn't have a .pan extension even if the current database does have a .pan extension. For example, suppose that you are working with a database named `Contacts.pan`. The procedure below will open the database named `Schedule.pan`.

```
openfile "Schedule"
```

However, what if the database you want to open is actually called `Schedule`, not `Schedule.pan`. In that case you must add the `nodefaultextension` statement immediately before the `openfile` statement, like this.

```
nodefaultextension
openfile "Schedule"
```

This revised procedure will open the `Schedule` database.

Name Extensions and Window Names

Panorama removes the .pan extension from the in-memory copy of the database. This means that you won't see the extension in the window title, and should not include the extension when using the `window` statement. Panorama also will not include the extension as part of the file name returned the `info("databasename")`, `info("windowname")`, or `info("files")` functions. You also should not include the extension in any statements or functions that require you to specify the name of an open database (for example `lookup()`, `grabdata()`, `arraybuild`, etc.) Bottom line -- just keep programming the same way you always have.

Flash Art Formulas

When the current database has a .pan extension and the Flash Art formula refers to a disk file (as opposed to a picture in the Flash Art Gallery or a resource) Panorama will automatically add the extension .pct to the final result (unless the formula generates an extension itself). You should make sure that any picture files used in a Flash Art or Super Flash Art formula end with the .pct extension (the Panorama Platform Converter will take care of this for you).

Using Partial Paths to Reference SubFolders

On the Macintosh you can use a "partial path" to reference a sub-folder of the folder that contains the database. Partial paths always begin with a colon. For example, `":Photos:Grand Canyon"` refers to a file named `Grand Canyon` in the `Photos` folder (the `Photos` folder must be in the same folder as the database). When this partial path is used on the PC, Panorama automatically converts the colons into backslashes for you. For example, you might use a partial path like this in a Flash Art SuperObject or in the `openfile` statement.

Hard Coded Folder Locations

If your program contains hard coded folder locations (for example `"My Disk:Samples:Contacts"`) these will have to be changed. Of course, you probably don't have any, since these would not work on different Macintosh systems either.

If you build a folder location with the `folderpath()` and `dbinfo()` functions, you'll still be alright. On the PC, this will result in a path that looks something like `C:\Samples\1999\`, which can be fed into the `folder()` function or used anywhere a path name may be used (for example Flash Art or the `openfile` statement).

The `info("panoramafolder")` function also works on both the Macintosh and the PC.

Is It a Mac or a PC?

Panorama V includes true-false functions for determining whether the current computer is running Windows, OS 9 or OS X.

Function	Description
windows()	True if running on a Windows computer
osx()	True if running on a Macintosh OS X computer
os9()	True if running on a Macintosh OS 9 computer

These functions are not available in earlier versions of Panorama. If you want your database to be compatible with Panorama 4 you can test for Macintosh vs. Windows this way:

```
if folderpath(info("panoramafolder"))[2,3]=":\ "  
    /* PC */  
else  
    /* Macintosh */  
endif
```

This works because all PC pathnames begin with a letter followed by :\
for example C:\
(main hard disk) or D:\
(cd-rom).